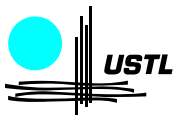
Numéro d'ordre: 3185



THÈSE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Raphaël MARVIE

Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants

Soutenue publiquement le 9 décembre 2002 devant la commission d'examen :

Président	:	Pr Mireille CLERBOUT	LIFL, Université de Lille I
Rapporteurs	:	Pr Jean BÉZIVIN	CRGNA, Université de Nantes
		Mr Keith DUDDY	DSTC, Brisbanne (Australie)
		Pr Michel RIVEILL	ESSI, Université de Nice
Co-rapporteur	:	Dr Mireille BLAY-FORNARINO	ESSI, Université de Nice
Examinateurs	:	Dr Laurent Rioux	Thales Research, Orsay
		Dr Sébastien Gérard	CEA, Saclay
Directeur	:	Pr Jean-Marc GEIB	LIFL, Université de Lille I
Co-directeur	:	Dr Philippe MERLE	INRIA, Université de Lille I

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL, URA 369 CNRS - Bât. M3 - UFR IEEA - 59655 VILLENEUVE D'ASCQ CEDEX
Tél. +33 (0) 3. 20. 43. 47. 21 - Fax. +33 (0) 3. 20. 43. 65. 66 - E-mail : marvie@lifl.fr

Au capitaine qui nous a laissé la barre tout en veillant du fond de nos cœurs...

Remerciements

Je tiens tout d'abord à remercier madame le professeur **Mireille Clerbout** pour m'avoir fait l'honneur d'accepter de présider le jury de cette thèse.

Je tiens ensuite à remercier messieurs les professeurs **Jean Bézivin** et **Michel Riveill**, monsieur **Keith Duddy** et madame **Mireille Blay-Fornarino** pour avoir accepté de rapporter cette thèse, ainsi que pour l'intérêt qu'ils ont porté à mon travail. Je les remercie aussi pour les discussions enrichissantes que nous avons pu avoir lors de nos différentes rencontres.

Enfin, je remercie messieurs **Laurent Rioux** et **Sébastien Gérard** d'avoir accepté de faire parti de mon jury.

Un immense et chaleureux merci au professeur **Jean-Marc Geib** pour m'avoir proposé un DEA puis cette thèse au sein de son équipe et « sous son aile ». Merci pour l'encadrement, les conseils, les encouragements, la confiance, les opportunités de rencontrer du monde... Un grand merci enfin pour nos divagations du vendredi soir qui transforment les fins d'après-midi en fins de soirées et donnent des perspectives à ne plus vouloir dormir. Ces remerciements vont aussi à **Philippe Merle** qui m'a co-encadré pendant ces trois années de thèse.

Un grand merci à **Christophe** « **tof** » **Gransart** sans qui rien de tout cela ne serait arrivé. Merci de m'avoir « monté » au troisième étage du M3 et ouvert les yeux sur ce qu'est la recherche académique. Merci aussi pour les encouragements et toutes nos discussions.

Une thèse se déroule dans une équipe de recherche, et quelle équipe ce fut dans mon cas. L'ambiance de GOAL a été un véritable plaisir et un élément moteur important. Merci à vous tous avec une « mention » particulière pour **Laurence Duchien** et toutes nos discussions, tes relectures et commentaires de la première version de ce document, ainsi qu'à **Bernard Carré** et **Gilles Vanwormhoudt** pour toutes nos discussions et pour l'approche à base de vues. Bernard encore pour les prolongations informelles et systématiques de tes cours de DEA.

Merci à tous les membres du projet RNRT CESURE, cadre dans lequel la première proposition CODeX a émergée, et de l'ARC SAMOA : Chantal, Daniel, Erik, Guy, Jean-Jacques, Lago, Marie-Pierre, Pierre, Olivier, Roland, Sarah et Vania.

Merci au reste du LIFL pour faire de cet endroit un cadre agréable de travail et de « loisirs ». Un grand merci tout particulier pour les encouragements à : David, Florence, Gilles (G.), Jean-Christophe, Mireille et Sophie. Enfin, merci à Caroline, Isabelle et Jean-Marc (T.) pour toutes vos relectures et vos commentaires.

Reste les nombreuses personnes, non citées par manque de place et sans doute un peu de mémoire, que j'ai eu la chance de croiser pendant ces trois années chargées de rencontres et qui ont su partager leurs points de vue et critiquer le mien, me faisant avancer dans ma réflexion.

Enfin, ces remerciements ne seraient pas complets sans une pensée toute particulière à ma famille et mes amis proches qui savent toujours être là quand et comme il le faut : Vous êtes merveilleux et je vous adore !

Support musical de la rédaction (par ordre alphabétique) : AC/DC (Live et Stuff uper Lip), Bach (Six suites pour violoncelle), Chet Baker (White Blues), Björk (l'intégrale), John Coltrane (Blue trane), Miles Davis ('Round about midnight et Kind of blue), Haendel (La grande sarabande), Massive Attack (Protection et Mezzanine), Mike Oldfield (l'intégrale), Metallica (S&M), Mozart (Requiem et Messe en ut mineur), Noir Désir (l'intégrale), Pink Floyd (The wall), Pixies (l'intégrale), Rachmaninov (Concerto n.3 pour piano), Joe Satriani (G3), Schubert (La jeune fille et la mort), The Strokes (This is it), Tchaikovsky (Concerto n.1 pour piano), Yann Tiersen (l'intégrale) et Vivaldi (Les quatre saisons et Concerto « La notte »).

Table des matières

1 Introduction **1**
- 1.1 Processus d'ingénierie du logiciel . . . 2
- 1.2 Acteurs du processus logiciel . . . 4
  - 1.2.1 Architecte logiciel . . . 4
  - 1.2.2 Développeur de composants . . . 5
  - 1.2.3 Intégrateur de composants . . . 6
  - 1.2.4 Placeur de composants . . . 6
  - 1.2.5 Déployeur d'applications . . . 7
  - 1.2.6 Administrateur d'applications . . . 8
  - 1.2.7 Synthèse . . . 8
- 1.3 Co-design dans le processus logiciel . . . 9
- 1.4 Composants et architectures . . . 10
  - 1.4.1 Définitions . . . 10
  - 1.4.2 Limitations . . . 12
- 1.5 Motivations et objectifs de notre proposition . . . 13
  - 1.5.1 Défis . . . 13
  - 1.5.2 Propositions . . . 14
- 1.6 Organisation de ce document . . . 14

I Etat de l'art **17**

2 Modèles de composants **19**
- 2.1 Un modèle abstrait de composants . . . 19
  - 2.1.1 Caractérisation des composants . . . 19
- 2.2 Modèles de composants académiques . . . 23
  - 2.2.1 Darwin . . . 23
  - 2.2.2 JavaPod . . . 26
- 2.3 Modèles de composants industriels . . . 28
  - 2.3.1 JavaBeans . . . 28
  - 2.3.2 Enterprise Java Beans . . . 31
  - 2.3.3 (D)COM/COM+ . . . 34
- 2.4 Modèles de référence . . . 37
  - 2.4.1 Open Distributed Processing . . . 37
  - 2.4.2 CORBA Component Model . . . 40
- 2.5 Conclusion . . . 44

3 Langages de description d'architectures 47
3.1 Concepts sous-jacents aux architectures . . . 47
3.1.1 Définition . . . 47
3.1.2 Composant . . . 48
3.1.3 Connecteur . . . 49
3.1.4 Configuration . . . 49
3.2 Langages formels de description d'architectures . . . 50
3.2.1 Rapide . . . 50
3.2.2 Wright . . . 52
3.3 Langages d'échange ou d'intégration d'architectures . . . 55
3.3.1 ACME . . . 55
3.3.2 xArch et xADL 2.0 . . . 57
3.4 Langages de configuration . . . 60
3.4.1 C2 . . . 60
3.4.2 Olan . . . 65
3.4.3 Descripteurs d'assemblages du CCM . . . 67
3.5 Conclusion . . . 70

4 Méta-modélisation, la vision de l'*Object Management Group* 75
4.1 Objectifs de la méta-modelisation . . . 76
4.1.1 Définitions . . . 76
4.1.2 De la nécessité de méta-modèles . . . 76
4.1.3 Mise en œuvre des techniques de méta-modélisation . . . 76
4.2 Le Meta Object Facility . . . 77
4.2.1 Présentation . . . 77
4.2.2 Le Modèle MOF . . . 79
4.2.3 Des modèles aux environnements . . . 82
4.2.4 Conclusion . . . 84
4.3 Séparation des préoccupations . . . 84
4.3.1 Approche . . . 84
4.3.2 Programmation orientée aspects . . . 85
4.3.3 Programmation structurée en contextes . . . 85
4.3.4 Séparation multi-dimentionelle des préoccupations . . . 85
4.4 *Model Driven Architecture* . . . 86
4.4.1 Fondements de la MDA . . . 87
4.4.2 Des modèles, des modèles, des modèles... . . . 88
4.4.3 Quelques moyens de mise en œuvre . . . 90
4.5 Conclusion . . . 91

II Séparation des préoccupations et méta-modélisation pour architectures logicielles 93

5 La proposition CODeX 95
5.1 Objectifs . . . 95

5.2 Vue d'ensemble de la proposition CODeX 97
5.2.1 Description d'architectures et séparation des préoccupations 97
5.2.2 Méthodologie de définition d'un ADL 101
5.3 CODeX et méta-modélisation 102
5.3.1 Méta-modélisation d'ADLs 102
5.3.2 La pile de méta-modélisation de CODeX 103
5.3.3 Support de la structuration des méta-modèles 104
5.3.4 Méta-méta-modèle de CODeX 106
5.4 Environnement associé à un ADL 107
5.4.1 Réification des architectures 107
5.4.2 De l'utilisation d'un référentiel de méta-données 109
5.4.3 Outillage associé aux référentiels 110
5.4.4 Chaîne de production des environnements 111
5.5 Conclusion 112

6 Méta-modélisation d'un ADL avec CODeX 115
6.1 Identification des préoccupations 115
6.1.1 Concepts communs à tous les acteurs 116
6.1.2 Mise en œuvre des composants 116
6.1.3 Placement des composants 116
6.1.4 Dynamique des applications 117
6.2 Définition du plan de base de l'architecture 117
6.2.1 Caractérisation des concepts 118
6.2.2 Méta-modèle du plan de base architectural 118
6.3 Définition de plans d'annotation 120
6.3.1 Plans d'annotation d'implémentation 121
6.3.2 Plans d'annotation de placement 123
6.3.3 Plans d'annotation de la dynamique 125
6.4 Définition d'un plan d'intégration 128
6.4.1 Définition des concepts 128
6.4.2 Méta-modèle du plan d'intégration 128
6.5 Conclusion 129

7 Production de l'environnement associé à un ADL 131
7.1 A propos de l'outillage MOF 131
7.1.1 *CODeX M2 tool* 132
7.2 Interfaces OMG IDL des référentiels 133
7.2.1 Projection en OMG IDL des packages MOF 133
7.2.2 Projection en OMG IDL des classes MOF 134
7.2.3 Projection en OMG IDL des associations MOF 136
7.3 Mise en œuvre des référentiels 139
7.3.1 Mise en œuvre des packages MOF dans le référentiel 139
7.3.2 Mise en œuvre des classes MOF dans le référentiel 142
7.3.3 Mise en œuvre des associations MOF dans le référentiel 143
7.3.4 Mise en œuvre des actions architecturales 145

7.4 Conclusion 146

8 Mise en œuvre de CODeX dans le cadre du CCM 149
8.1 Introduction 149
8.2 Environnement pour le *CORBA Component Model* 150
8.2.1 Spécialisation du référentiel 150
8.3 Support à la mise en œuvre des applications 155
8.3.1 Règles de projection vers le langage OMG IDL 3 155
8.3.2 Production des interfaces OMG IDL 3 158
8.4 Dynamique des applications 159
8.4.1 Mise en œuvre des actions architecturales 159
8.4.2 Discussion sur la mise en œuvre du déploiement 161
8.5 Utilisation de l'environnement 163
8.5.1 Définition de l'architecture 163
8.5.2 Définition des interfaces de mise en œuvre 165
8.5.3 Spécification du placement et des archives 165
8.5.4 Définition de l'action de déploiement 167
8.6 Conclusion 169

III Conclusion et perspectives 171

9 Conclusion et perspectives 173
9.1 Travaux réalisés 173
9.1.1 Méthodologie 174
9.1.2 Outillage 175
9.2 Perspectives 176

A Publications 179

B Acronymes 183

Table des figures

1.1 Modèles du processus unifié 3
1.2 Rôle et moyens associés aux acteurs du processus logiciel 9
1.3 Conception collaborative d'applications 10

2.1 Méta-modèle d'une caractérisation des composants logiciels 20
2.2 Description d'un composant avec Darwin 24
2.3 Définition d'un composant composite avec Darwin 25
2.4 Représentation graphique avec Darwin 25
2.5 Modèle d'un composant JavaPod 27
2.6 Architecture de la plate-forme JavaPod 27
2.7 Modèle de composant JavaBean 29
2.8 Définition d'un JavaBean 30
2.9 Modèle abstrait des Enterprise Java Beans 31
2.10 Définition d'un Enterprise Java Bean et de sa maison 32
2.11 Environnement d'exécution des Enterprise Java Beans 33
2.12 Modèle binaire des composants COM 34
2.13 Composition (a) et agrégation (b) avec COM 36
2.14 Modèle de composants RM-ODP 39
2.15 Eléments du modèle d'ingénierie d'ODP 40
2.16 Modèle abstrait du CORBA Component Model 41
2.17 Définition d'un type de composant CORBA et d'un type de maison 42
2.18 Synthèse des concepts présents dans les modèles de composants 44

3.1 Exemple de définition d'architecture avec *Rapide* 51
3.2 Exemple de définition d'architecture avec *Wright* 53
3.3 Exemple de définition d'architecture avec *ACME* (extrait) 56
3.4 Exemple de définition d'un type de composant avec *xADL* 59
3.5 Exemple de définition d'un type de composant avec *C2* 62
3.6 Exemple de définition d'architecture avec le style *C2* 63
3.7 Exemple de modification dynamique d'architecture avec le style *C2* 64
3.8 Exemple de relation architecture / implantation avec le style *C2* 64
3.9 Description d'un composant avec Olan 66
3.10 Représentation graphique de composants avec Olan 66
3.11 Composant composite dans Olan 66
3.12 Attributs de déploiement dans Olan 67
3.13 Exemple d'assemblage de composants pour le CCM (extrait) 69
3.14 Synthèse des préoccupations et concepts présents dans chaque ADL 71

4.1 Architecture à quatre niveaux du MOF 78
4.2 Méta-modèle schématique de la MDA 89

5.1 Comparaison entre ADLs et niveaux d'abstraction de CODeX 98
5.2 Principe d'enrichissement du plan de base architectural 99
5.3 Pile de méta-modélisation à quatre niveaux de CODeX 103
5.4 Méta-méta-modèle de CODeX 107
5.5 Vues par préoccupation du référentiel de méta informations 108
5.6 De la définition à l'utilisation d'un ADL avec CODeX 111

6.1 Plan de base architectural pour applications réparties 119
6.2 Plan de base d'annotation pour les implémentations de composants 122
6.3 Plan d'annotation pour les implémentations de composants 123
6.4 Plan de base d'annotation pour le placement des composants 124
6.5 Plan d'annotation pour le placement des composants 125
6.6 Plan de base pour la dynamique des architectures 127
6.7 Plan d'annotation pour la dynamique des architectures 127
6.8 Définition du package d'intégration par héritage 128
6.9 Extrait du package d'intégration des préoccupations 129

7.1 Projection du package *ArchitecturalBase* en OMG IDL 133
7.2 Projection du package *Location* en OMG IDL 134
7.3 Projection de la classe *ComponentDef* en OMG IDL 135
7.4 Projection de l'association *Provides* en OMG IDL 137
7.5 Projection de l'association *RunsOn* en OMG IDL 138
7.6 Implémentation du package *ArchitecturalBase* en OMG IDLscript (extrait) . 140
7.7 Implémentation du package d'annotation *Location* en OMG IDLscript 141
7.8 Implémentation du package d'intégration *MyADL* en OMG IDLscript 141
7.9 Gestion de l'héritage multiple lors de l'initialisation des classes de packages . 142
7.10 Implémentation de la classe `ComponentDef` du package `ArchitecturalBase` en OMG IDLscript 143
7.11 Implémentation de la classe d'association `Provides` du package `ArchitecturalBase` en OMG IDLscript (extrait) 144
7.12 Implémentation de la classe `ActionDef` du package `DynamismBase` en OMG IDLscript (extrait) 145

8.1 Organisation de la représentation des applications 150
8.2 Spécialisation du concept de composant primitif pour le CCM 152
8.3 Spécialisation de l'interface `PrimitiveDef` pour le CCM 152
8.4 Spécialisation du concept de composite pour le CCM 153
8.5 Projection de l'interface `CompositeDef` pour le CCM 153
8.6 Mise en œuvre de l'interface `PrimitiveDefCCM` en IDLscript 155
8.7 Projection de la définition d'un philosophe en OMG IDL 3 156
8.8 Projection d'un connecteur incluant des traitements en OMG IDL 3 157
8.9 Projection des ports d'un composite en OMG IDL 3 158

8.10 Mise en œuvre de l'opération `create` pour le CCM (extrait) 160
8.11 Mise en œuvre de l'opération `OpBind` pour le CCM 161
8.12 Définition du composite `diner` et d'un primitif `philosophe1` 163
8.13 Définition du port `main_gauche` du composant `philosophe1` 164
8.14 Définition d'un *connecteur* entre deux *primitifs* 164
8.15 Projection en OMG IDL 3 des interfaces de *primitifs* 165
8.16 Mise en relation du primitif `philosophe1` avec son implantation 166
8.17 Mise en relation du primitif `philosophe1` avec son serveur d'exécution 167
8.18 Définition de l'opération `deploy` sur le composite `diner` 168

Chapitre 1

Introduction

L'utilisation de composants logiciels pour réaliser des applications a fait évoluer une partie de cette activité de la programmation vers l'assemblage de briques logicielles. L'expression de cet assemblage, l'architecture, devient un élément central dans la réalisation des applications. De plus, la production d'applications se rationalise et suit de plus en plus une démarche industrielle au travers de processus d'ingénierie du logiciel (ou processus logiciels). Ces processus définissent un ensemble de rôles pouvant être joués par des personnes différentes. Il en résulte qu'une architecture logicielle est destinée à être partagée par un ensemble d'acteurs afin de collaborer à la réalisation de tâches complémentaires visant à construire une application.

Les langages de description d'architectures (ADL, *Architecture Description Languages*) représentent actuellement le moyen de définition et d'exploitation des architectures logicielles. Ces langages permettent de préciser les composants constituant une application, leur interconnexion et leurs propriétés. Cependant, les langages disponibles sont peu structurés, les informations relatives à une architecture sont mélangées à un seul niveau, bien souvent syntaxique. Il en résulte que la collaboration des différents acteurs autour des architectures n'est pas facilitée. D'autre part, ces langages offrent des moyens figés pour exprimer les éléments d'une architecture logicielle. Ils ne sont donc pas toujours en adéquation avec les besoins des acteurs du processus logiciel : absence de support à certains besoins ou moyen inadapté pour exprimer certains éléments de l'architecture.

L'objectif de notre travail est de proposer une solution à ces limitations des langages de description des architectures. Dans un premier temps, ce travail vise à fournir un environnement structuré de manipulation cooperative des architectures. La collaboration des acteurs autour de l'architecture devient facilitée pendant le processus logiciel. Ensuite, afin de répondre au mieux à des besoins spécifiques, notre proposition ne vise pas à définir un environnement généraliste, mais un ensemble de moyens pour définir et produire l'environnement le plus adapté à un processus logiciel et un ensemble d'acteurs donnés. Notre proposition se place donc à un niveau supérieur aux ADLs pour permettre de définir des ADLs structurés et sur mesure afin de construire des applications à base de composants logiciels. Le dernier objectif de notre approche est de ne pas se limiter à une forme syntaxique de la représentation des architectures, mais d'en fournir une forme réifiée, et donc manipulable plus simplement. Afin d'atteindre ces objectifs, notre approche repose sur l'utilisation des techniques de métamodélisation et sur la mise en œuvre de la séparation des préoccupations.

Cette introduction est organisée comme suit.

– La production d'applications fait intervenir un certain nombre d'acteurs collaborant dans le cadre d'un processus logiciel. La section 1.1 présente ce qu'est un processus logiciel.

- La section 1.2 présente le découpage du processus logiciel que nous utilisons comme support tout au long de ce document, et les besoins qui s'en dégagent.
- La section 1.3 discute brièvement des besoins sous-jacents à la collaboration de ces différents acteurs.
- La section 1.4 présente rapidement les deux solution actuelles les plus adaptées pour répondre aux besoins des acteurs d'un processus logiciel. Elle souligne aussi leur limitations.
- Sur la base de ces limitations, la section 1.5 présente les motivations et objectifs de ce travail.
- Enfin, la section 1.6 présente l'organisation de de document.

1.1 Processus d'ingénierie du logiciel

Dans leur livre discutant du processus unifié de développement logiciel [39] (RUP *Rational Unified Process*), Jacobson, Booch et Rumbaugh répondent à la question « Qu'est ce qu'un processus de développement logiciel? » de la manière suivante.

> Un processus définit qui fait quoi, à quel moment et de quelle façon pour atteindre un certain objectif. Dans le domaine de l'ingénierie logicielle, le but consiste à élaborer un produit logiciel ou à en améliorer un existant. Un processus digne de ce nom doit fournir des directives garantissant le développement efficace de logiciels de qualité et présenter un ensemble de bonnes pratiques autorisées par l'état de l'art. Ces dispositions permettent de réduire les risques tout en améliorant la prévisibilité. Il s'agit, d'un point de vue plus général, de promouvoir une vision et une culture communes. [...]

Le *processus unifié* tel que défini par Jacobson, Booch et Rumbaugh est un processus de développement logiciel. Il vise donc à transformer les besoins des usagers en applicatifs logiciels. Au delà d'un simple processus figé, il représente un *framework* de processus générique. Il a donc comme motivation sous jacente d'être appliqué en fonction du contexte dans lequel il est utilisé. Le fait que le processus unifié commence avec les besoins des usagers motive l'utilisation de cas d'utilisation. Un *cas d'utilisation* représente une fonctionnalité de l'applicatif offerte soit à un utilisateur final soit à un autre applicatif. L'ensemble des besoins fonctionnels définit le modèle des cas d'utilisation, ce qui était classiquement appelé spécification fonctionnelle de l'applicatif. Ce modèle des cas d'utilisation sert de base à la conception à proprement parlé des applications.

Le cœur du processus unifié est l'architecture. *L'architecture logicielle* définit les aspects statiques et dynamiques d'une application. Elle résulte des besoins définis dans le paragraphe précédent et découle directement des cas d'utilisation. Toutefois, la définition de l'architecture est influencée par des facteurs externes aux cas d'utilisation comme la plateforme d'exécution, les briques logicielles existantes, les contraintes de déploiement, ou encore les besoins non fonctionnels (tolérance aux pannes, performance, etc). L'architecture propose une vue d'ensemble de la conception d'une application reflétant les caractéristiques voulues tout en laissant de côté les détails secondaires. Jacobson, Booch et Rumbaugh présentent

les cas d'utilisation comme la fonction et l'architecture comme la forme. Le processus unifié encourage à ce que les deux évoluent de façon concomitante.

Enfin, le processus unifié est défini comme itératif et incrémental. Il encourage le découpage d'un projet en plusieurs parties, les *itérations*, qui correspondent à une étape d'enchaînement d'activités. Les *incréments* représentent le résultat de chaque itération, ce sont les stades de développement du produit final, *i.e.* l'application. Une itération prend en compte un certain nombre de cas d'utilisation, ce qui tend à améliorer l'utilisabilité de l'application et la réutilisation de sa définition. Chaque itération est indépendante dans le sens où elle correspond à un mini projet qui part de cas d'utilisations pour aboutir à du code exécutable. On retrouve alors le cycle « analyse, conception, implantation et test » à une échelle réduite. Un incrément représente alors un ajout de fonctionnalités ou simplement une restructation / amélioration de fonctionnalités existantes.

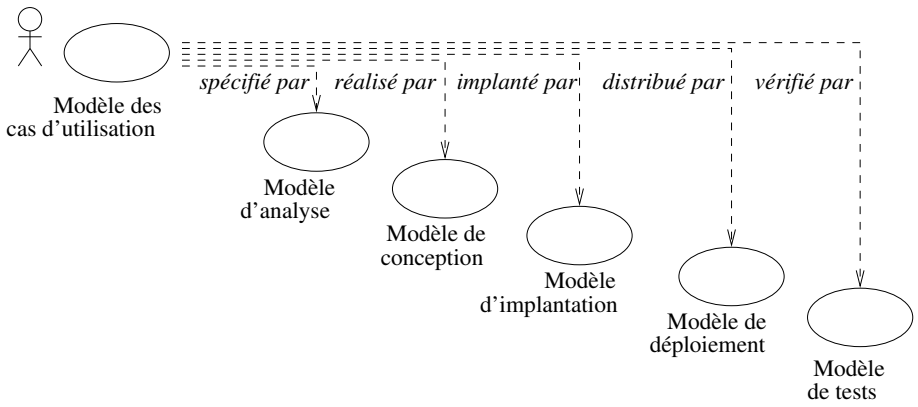


FIG. 1.1 – *Modèles du processus unifié*

Le processus unifié répète donc un certain nombre de fois une série de cycles qui mis bout à bout représentent le cycle de production de l'application. Dans ce contexte, chaque cycle correspond à la production d'une nouvelle version de l'application. La production des briques de l'application, les composants, n'est pas une fin en soi. Pour ce fonctionnement en cycles, **un ensemble d'informations doivent leur être associées et rendues disponibles**. Le processus unifié incite donc à bien définir et à utiliser un certain nombre de modèles représentant l'application. Ces modèles sont illustrés dans la figure 1.1. Ils sont liés entre eux et représentent l'application comme un tout. Les éléments de ces différents modèles présentent les éléments de traçabilité permettant de passer d'un modèle à un autre au cours d'un cyle. Ils sont définis comme suit :

- le modèle des cas d'utilisation décrit à la fois les cas d'utilisation et leurs relations avec les usagers (utilisateurs finaux et applications tierces) ;
- le modèle d'analyse détaille les cas d'utilisation et spécifie une première répartition du comportement de l'application en différentes entités ;
- le modèle de conception définit la structure statique du système sous forme de contrats de composants ainsi que les collaborations entre ceux-ci ;
- le modèle d'implantation intègre les composants et leurs implantations, la manière dont

ils sont réalisés ;
- le modèle de déploiement définit les noeuds du système sous jacent et l'affectation des composants sur les noeuds ;
- le modèle de test décrit les cas de tests vérifiant les cas d'utilisation.

Tous ces modèles représentent une masse d'information qui forme ce que l'on appelle l'architecture dans toutes ses dimensions.

1.2 Acteurs du processus logiciel

Le RUP (*Rational Unified Process*) représente un guide à suivre pour définir le processus d'ingénierie du logiciel le plus adéquat au contexte applicatif considéré. Les acteurs intervenant dans de tels processus peuvent fortement varier d'un contexte à un autre. Dans le cadre des applications à base de composants distribués, nous considérons six acteurs principaux : architecte logiciel, développeur de composants, intégrateur de composants, placeur de composants, déployeur d'applications et administrateur d'applications ; détaillés ci-après.

En plus de ces six acteurs, des spécialistes de domaines particuliers interviendront certainement dans un processus logiciel. Leurs rôles pouvant varier d'un contexte à un autre ils ne seront pas présentés ici. Nos préoccupations premières ne sont pas liées aux propriétés non fonctionnelles mais architecturales. La persistance, les transactions, la sécurité sont des exemples de domaines pour lesquels des spécialistes seront parfois requis. Enfin, nous n'avons pas isolé un acteur sous l'étiquette *analyste*. Nous considérons que la tâche de chaque acteur présenté ici inclut une part d'analyse. Il en est de même pour le test dans le sens où chaque acteur doit à son niveau effectuer le test des éléments qu'il produit. Nous sommes aussi conscients que ce découpage pourrait être plus fin. Cependant un tel découpage n'apporterait pas d'éléments supplémentaires à notre discours, et nous essayons de conserver un cadre de travail relativement synthétique.

Nous présentons dans cette section un découpage du processus logiciel. Ce découpage illustre pour chaque participant, son ou ses rôles et les besoins induits, c'est-à-dire les outils et moyens nécessaires à un acteur pour effectuer sa tâche.

1.2.1 Architecte logiciel

1.2.1.1 Rôle

Le rôle de l'architecte logiciel est tout d'abord de définir le cœur d'une application, c'est-à-dire sa structure. Dans le contexte des composants logiciels, une application répartie est définie par l'ensemble de ses composants, fournissant les briques de base, ainsi que par leurs interconnexions, définissant la structure de l'application. Un architecte doit donc définir d'une part, les contrats des composants requis par une application et, d'autre part, au travers de quelles interfaces deux composants sont mis en relation. De plus, il est important de définir l'état abstrait du composant, qui servira par la suite à lui assigner une configuration initiale. Cette partie servira de base à la réalisation et à l'instanciation d'un assemblage de composants, c'est-à-dire une instance de l'application.

La seconde activité d'un architecte est de définir la dynamique de l'application, c'est-à-dire les évolutions possibles de sa structure au cours de son exécution. La définition de

la dynamique d'une application permet sa reconfiguration à l'exécution. La dynamique est essentiellement exprimée en termes de composants à ajouter ou à supprimer, ainsi qu'en l'établissement et la suppression de connexions entre composants. Ces modifications influent sur la structure et le comportement des applications.

L'entrée d'un nouvel usager final dans une application illustre bien le besoin de dynamique des architectures. Pour que l'usager puisse utiliser les fonctionnalités de l'application, il est nécessaire de lui déployer une partie cliente à connecter à l'application déjà déployée. De nouveaux composants vont donc être déployés, configurés puis connectés entre eux et avec des composants existants de l'application.

1.2.1.2 Moyens

Pour produire l'architecture de base d'une application, un architecte doit disposer d'un, ou de plusieurs, outil(s). Indépendamment du choix d'un type d'outil particulier, il est primordial que l'expression de l'architecture de base soit exploitable par les autres acteurs du processus d'ingénierie du logiciel. La fourniture d'une version manipulable de l'architecture de base facilite la contribution des différents acteurs à la définition de l'architecture globale de l'application. Un format standard de représentation des architectures est un pré-requis de tout processus logiciel. Il contribue à la collaboration des différents acteurs.

1.2.2 Développeur de composants

1.2.2.1 Rôle

Un développeur de composants logiciels a pour objectif de produire les implantations des composants définis par un architecte, en respectant les interfaces spécifiées par ce dernier. Comme les applications réparties tendent à s'exécuter dans un contexte hétérogène, il pourra être nécessaire au développeur de composants de fournir plusieurs implantations d'un même composant pour des utilisations dans des environnements différents. Cela peut signifier d'une part que pour une technologie de composant donnée, ces implantations soient réalisées dans différents langages, et d'autre part, que des implantations soient réalisées dans le cadre de technologies différentes. Une fois les implantations réalisées, le développeur de composants devra en fournir des versions exécutables avec leurs descriptions (mode d'emploi) regroupées dans des archives diffusables.

1.2.2.2 Moyens

Afin de produire des archives de composants utilisables, un développeur aura besoin de plusieurs éléments. Premièrement, il doit avoir accès à la définition des composants produits par un architecte qu'il doit implanter, c'est-à-dire à l'ensemble des interfaces d'un composant. Ensuite, il est souhaitable que l'environnement de développement utilisé inclut une part de génération automatique de code à partir des contrats produits par les architectes. Cette génération de code doit tendre à produire les parties non fonctionnelles d'un composant, comme la gestion de la connectique. Il ne reste alors au développeur qu'à produire la logique métier du composant, c'est-à-dire la partie fonctionnelle. Enfin, il doit disposer d'un outil

de création d'archives contenant idéalement à la fois les implantations, les interfaces et les descriptions des composants.

1.2.3 Intégrateur de composants

1.2.3.1 Rôle

Ce que nous définissons comme l'intégration de composants est la mise en relation des composants définis par l'architecte et de leurs implantations réalisées par les développeurs. Dans le cas où les archives de composants ne sont pas mises en ligne par les développeurs mais simplement fournies, l'intégrateur doit les rendre disponibles pour le chargement automatique à distance dans le but de supporter le téléchargement au moment de la phase de déploiement des applications. Cette mise à disposition repose sur l'utilisation de référentiels d'archives, *i.e.* des étagères électroniques de composants.

Ensuite, l'intégrateur enrichit l'architecture de base avec l'information de localisation des archives de composants à utiliser. Cette enrichissement peut se faire de deux façons : un composant est associé à la localisation précise d'une archive ou aux caractéristiques de son implantation. Dans le second cas, une fonction de courtage est utilisée au moment du déploiement pour choisir l'implantation la plus appropriée (dont les caractéristiques correspondent) pour le site d'exécution considéré.

1.2.3.2 Moyens

L'intégrateur doit disposer au moins de deux outils. En premier lieu, il a besoin d'un outil de gestion des référentiels d'archives. Cet outil doit permettre d'une part, d'ajouter ou de retirer des archives et, d'autre part, d'offrir la navigation au sein du référentiel. Il est alors possible de connaître la liste des archives disponibles pour un référentiel donné ou de rechercher des archives disponibles en fonction, par exemple, des caractéristiques de leur contenu. En second lieu, l'intégrateur de composants doit disposer d'un outil de manipulation de l'architecture. Cet outil doit lui permettre de manipuler l'architecture de base, mais surtout d'enrichir celle-ci avec les informations relatives aux archives de composants à utiliser.

1.2.4 Placeur de composants

1.2.4.1 Rôle

L'acteur que nous définissons comme *placeur* de composants est associé à deux rôles complémentaires. En premier lieu, il définit la représentation des sites d'exécution de composants. En particulier, il spécifie les capacités d'exécution de ces sites. Par exemple les types de binaires acceptés par ce site : architecture Intel ou Sparc, langage Java ou C++, *etc.* Dans le cas d'applications ayant des contraintes de qualité de service, il peut aussi être nécessaire de représenter les capacités de traitement du site : la mémoire, la ressource CPU ou encore la charge de la machine. De manière similaire, l'interconnexion des sites d'exécution doit être exprimée. Pour cela le type de réseau connectant deux sites d'exécution doit être précisé. Ici encore, dans un contexte de qualité de service, les capacités du réseau peuvent être à préciser : bande passante garantie, débit maximal, *etc.*

En second lieu, le placeur de composants associe les composants spécifiés par l'architecte avec les sites sur lesquels ceux-ci vont s'exécuter. Tout comme l'intégrateur, il va enrichir l'architecture de base, mais ici avec des informations relatives à l'exécution des composants. Tout comme l'intégrateur, le placeur a deux options : il associe un composant de l'architecture avec un site d'exécution ou avec une caractérisation du site et de ses interconnexions (réseau) sur lequel il doit s'exécuter. Ici encore, une fonction de courtage est utilisée au moment du déploiement pour trouver un site adéquat et y déployer une archive de composants.

1.2.4.2 Moyens

Le placeur doit, lui aussi, disposer de deux types d'outils. En premier lieu, il doit disposer d'un moyen de configurer un site d'exécution. Dans le cadre des technologies à base de composants, ceci se traduit par la mise en place et la configuration de serveurs de composants. Ces derniers ont pour but de permettre le déploiement d'implémentations à la demande, ainsi que de servir de support à l'instanciation et à l'exécution des composants. En second lieu, toujours de manière similaire à l'intégrateur, le placeur doit disposer d'un outil de manipulation de l'architecture de base. Cet outil doit lui permettre d'accéder à la définition de cette architecture pour l'enrichir des informations relatives aux sites d'exécution et pour créer les associations entre les composants et leurs sites d'exécution.

1.2.5 Déployeur d'applications

1.2.5.1 Rôle

Le déploiement d'applications à base de composants distribués peut être vu à deux niveaux : premièrement le déploiement des composants, c'est-à-dire la mise à disposition des archives sur les sites d'exécution, leurs instanciations et leurs configurations de base ; ensuite, l'interconnexion de ces instances de composants et l'application d'une configuration plus globale. Certains modèles de composants donnent des directives pour réaliser le déploiement. Cependant, ces directives ne sont pas nécessairement mises en œuvre au sein des outils associés : de plus, un processus de déploiement générique ne convient pas forcément tel quel à toutes les applications qui peuvent avoir des besoins spécifiques.

Pour ces deux raisons, le déployeur d'applications définit ou précise comment le déploiement d'une application donnée doit être réalisé. Il peut, par exemple, avoir à préciser dans quel ordre les composants doivent être déployés et interconnectés. Tout comme les deux acteurs précédents, le déployeur d'application va enrichir la définition de l'architecture pour y intégrer la définition du processus de déploiement. La seconde activité du déployeur d'application est de réaliser le déploiement. Il va, pour ce faire, exploiter la définition qu'il a produite et s'assurer de son bon déroulement. La dynamique de l'application, c'est-à-dire sa reconfiguration, peut aussi être précisée. La reconfiguration peut être perçue comme un redéploiement partiel d'une application : c'est le cas, par exemple, de l'insertion d'un nouvel usager final.

1.2.5.2 Moyens

A la différence des deux acteurs précédents, un déployeur ne doit pas seulement avoir accès à l'architecture de base telle que définie par l'architecte, mais à sa version enrichie par

l'intégrateur et le placeur. En effet, le déploiement ne repose pas uniquement sur l'assemblage des composants, mais aussi sur « quoi » déployer, résultat de la contribution d'un intégrateur, et « où » déployer, résultat de la contribution d'un placeur.

En termes d'outils, les besoins sont similaires aux acteurs précédents. Un outil de manipulation est nécessaire pour qu'un déployeur accède à une architecture et puisse l'enrichir du processus de déploiement. Ensuite, le déployeur doit disposer d'un outil qui va réaliser le déploiement d'une application en se fondant sur la définition de l'architecture résultante. Cet outil doit permettre une automatisation de l'évaluation du processus tout en fournissant une certaine interactivité pour permettre au déployeur de superviser cette évaluation et éventuellement de l'influencer.

1.2.6 Administrateur d'applications

1.2.6.1 Rôle

Dans le cadre des applications réparties, l'administration est un réel problème, qui plus est dans le cas des applications devant assurer une disponibilité 24h/24. De plus, le temps d'exécution d'une telle application est très supérieur au temps de production. L'administrateur d'applications a comme rôle de superviser et de reconfigurer, lorsque cela est nécessaire, une application répartie. L'activité de supervision requiert à la fois une vision de l'exécution d'une application, la localisation des composants et leur interconnexion, ainsi que la connaissance de son état et de son comportement.

Notre exemple de l'insertion d'un nouvel usager final au sein de l'application représente un cas où l'administrateur devra déclencher l'évaluation d'un redéploiement, au moins partiel, de l'application. Dans ce type de cas, l'administrateur exploite des opérations qui ont été définies et rendues disponibles par les acteurs précédents du processus d'ingénierie du logiciel, et plus précisément par le déployeur.

1.2.6.2 Moyens

Dans le but de disposer d'une vision globale d'une application, l'administrateur devra disposer de l'architecture complète de cette application. C'est sur la base de l'expression de cette architecture qu'il pourra retrouver les différents constituants d'une application. En plus de la vision d'ensemble, l'architecture, telle que nous l'avons présentée jusqu'ici, fournit à l'administrateur le jeu d'actions disponibles pour reconfigurer l'application. Contrairement aux intégrateurs et aux placeurs, un administrateur ne va *a priori* pas modifier une architecture ; il lui suffit donc de disposer d'un outil de visualisation de celle-ci. Toutefois, cet outil doit permettre d'agir en parallèle sur l'application : il doit permettre d'invoquer les actions de reconfiguration. Une bonne solution serait un outil graphique fournissant les informations relatives aux sites d'exécution, aux connexions réseaux, aux composants et aux assemblages.

1.2.7 Synthèse

Le tableau de la figure 1.2 synthétise les acteurs du processus logiciel. Il exprime pour chacun d'entre eux les informations utilisées et produites, ainsi que les moyens requis. Ces in-

formations et moyens devraient se retrouver dans un environnement ou langage de description d'architectures.

Acteur	Utilise	Fournit	Moyens
Architecte	-	Architecture de base	Outil de definition d'architectures
Développeur	Architecture de base	Implantations de composants	Outil de production d'implantations composants
Intégrateur	Architecture de base, implantations de composants	Archives de composants, intégration dans l'architecture	Outil d'archivage, outil de définition d'architecture
Placeur	Architecture de base	Serveurs de composants, placement	Environnement d'exécution de composants, outil de définition d'architecture
Deployeur	Architecture complète	Processus de déploiement, instance d'application	Outil de déploiement, Environnement d'exécution
Administrateur	Architecture complète, instance d'application	Maintenance	Console, Environnement d'exécution

FIG. 1.2 – *Rôle et moyens associés aux acteurs du processus logiciel*

1.3 Co-design dans le processus logiciel

Le découpage du processus logiciel retenu dans le cadre de ce travail illustre l'aspect central de l'architecture d'une application. Il met aussi en avant le fait que chaque acteur a des préoccupations différentes vis-à-vis de l'architecture, aussi bien en termes d'utilisation que de contribution. Cependant, le processus d'ingénierie du logiciel vise la production d'une application, ce qui implique un but commun pour les différents acteurs. La collaboration est donc une nécessité. Une fleur est une bonne illustration de la collaboration des acteurs autour de l'architecture d'une application (voir la figure 1.3). Le cœur de la fleur représente la définition de l'architecture, et les pétales les différents acteurs qui gravitent autour de cette définition. Les interactions se font toujours entre les pétales et le cœur de la fleur : les éléments architecturaux utilisés et produits par un acteur.

Afin de permettre la collaboration des différents acteurs, deux éléments sont importants. Premièrement, chaque acteur doit disposer d'un cadre de travail bien défini, d'une méthodologie, qui lui permettra de contribuer à la production des applications sans risquer de sortir du cadre général. Sans ce contrôle, produire des éléments incompatibles est un risque latent. En second lieu, nous avons pu observer dans la description de nos différents

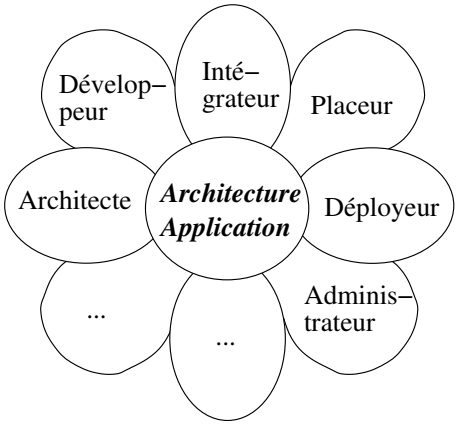


FIG. 1.3 – *Conception collaborative d'applications*

acteurs qu'ils partagent un certain nombre d'informations. Par exemple, l'architecte fournit la définition des composants aux intégrateurs et aux placeurs. Il est donc nécessaire de définir un formalisme de partage de ces informations entre tous les acteurs du processus. Cela permet à chacun d'exploiter les informations produites par ses prédécesseurs et de fournir des informations utilisables par ses successeurs.

Résumé. Un format standard de description des architectures doit être partagé entre les différents acteurs d'un processus logiciel. Une méthodologie d'utilisation de ce format et les outils associés doivent fournir un cadre de travail aux acteurs.

1.4 Composants et architectures

Les composants logiciels et les langages de description d'architectures (ADLs) sont deux moyens utilisés actuellement pour répondre aux besoins des processus logiciels. Après avoir rappelé leurs définitions, le lien entre ces deux approches et leurs limitations sont présentés.

1.4.1 Définitions

1.4.1.1 Composant logiciel

Il n'existe pas, aujourd'hui, *une* définition du terme « composant logiciel. » Toutefois, plusieurs définitions ou caractérisations sont bien acceptées. Jed Harris, président du CI Lab, a donné la définition suivante d'un composant, en 1995 :

> *Un composant est un morceau de logiciel assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour que l'on puisse l'installer et en assurer le support. De plus, il est doté d'interfaces standards pour pouvoir interopérer.*

Ensuite, lors de la première édition du *Workshop on Component Oriented Programing* (1996) [91], les participants ont convenu de la définition suivante :

> *A software component is a unit of composition with contractually specified inter-*

faces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Ces deux définitions soulignent l'objectif premier de l'utilisation des composants logiciels. Le souhait est de disposer, plutôt que de reproduire régulièrement, des briques de base nécessaires à la construction des applications. L'activité de construction devient ainsi une activité d'assemblage, et ceci pour sa simplification et sa rationalisation. Les besoins évoluent donc de supports à la programmation vers des supports à la description. Ce besoin de description est à l'intersection de l'approche composant et des travaux liés aux architectures logicielles.

1.4.1.2 Architecture logicielle

Dans [87], Mary Shaw *et al.* définissent une architecture logicielle de la manière suivante :

The architecture of a software system defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, and component compatibility. Architectural models clarify structural and semantic differences among components and interactions. Architectural definitions can be composed to define larger systems. [...]

Le parallèle avec l'industrie du bâtiment est donc direct. Lors de la construction d'un bâtiment, des composants de base comme les portes, les fenêtres, les murs sont assemblés en suivant un certain nombre de règles architecturales. Un architecte définit un ensemble de plans décrivant cet assemblage. Lors de la construction d'une application, l'idée est d'utiliser des composants existants et de les assembler en suivant un certain nombre de règles. La description de l'architecture de l'application va exprimer comment ces composants doivent être organisés pour fournir les traitements requis.

1.4.1.3 Lien entre les deux approches

Bien qu'issus de deux communautés, les travaux relatifs aux composants et aux architectures logiciels sont complémentaires. Les premiers visent à fournir des briques de base de qualité, réutilisables pour la production d'applications. Les seconds visent à l'intégration de ces briques à un niveau plus global pour construire des applications de qualité, maintenables. Dans les travaux des deux communautés, les concepts manipulés sont similaires, seul le niveau auquel ils sont manipulés varie. Dans le cadre d'un modèle de composants, l'accent est essentiellement mis sur la manière de définir un composant avec les propriétés voulues et bien agencées. La partie relative à l'assemblage de ces composants est souvent assez pauvre. Dans le cas des langages de description d'architectures (ADL *Architecture Description Languages*), la façon d'obtenir les composants est secondaire, alors que les propriétés de leurs interactions sont étudiées en détail.

Dans les deux cas, l'objectif final est identique : produire rapidement des applications de qualité, facilement maintenables et évolutives. Les deux propositions tendent à faciliter la construction d'applications en réutilisant au mieux les éléments déjà disponibles, composants

ou éléments architecturaux. Les modèles de composants et les ADLs sont ainsi des supports significatifs au déroulement d'un processus logiciel. Ils contribuent, chacun à leur niveau, aux besoins des différents acteurs du processus. L'architecte utilise un ADL pour définir l'architecture de l'application et utilise un modèle de composants pour spécifier les éléments de base. Un développeur va produire des composants dans le cadre d'un modèle technologique particulier. Un intégrateur va enrichir l'architecture avec les mises en œuvre de composants à utiliser pour une instance précise de l'application. Et ainsi de suite pour les autres acteurs.

1.4.2 Limitations

Les deux approches ont, de par leur objectif propre, des limitations intrinsèques. En règle générale, les modèles de composants ont un pouvoir de description des assemblages assez limité. La spécification d'une architecture se réduit à exprimer l'interconnexion des composants. Très peu de propriétés peuvent être exprimées sur l'ensemble de l'architecture, seules les propriétés des composants le sont. Du côté des ADLs, les capacités des modèles de composants sont peu utilisées. Les modèles de composants utilisés sont relativement simples et ne bénéficient pas des travaux centrés sur les composants. Les environnements d'exécution de ces composants ne sont que peu voire pas du tout spécifiés. **Les points forts des modèles de composants et des ADLs ne sont pas actuellement exploitables simultanéments au sein d'un processus logiciel.**

Non content de ne pouvoir bénéficier simultannément des atouts des ADLs et des modèles de composants, le choix d'un ADL impose une solution technologique sous-jacente. La définition de l'architecture d'une application est dépendante de cette solution technologique. L'évolution de la mise en œuvre d'une architecture, par exemple vers un nouveau modèle technologique de composants, impose une reconception de celle-ci pour ce nouveau modèle. Il n'est pas possible de capitaliser l'expression d'une architecture en vue de sa mise en œuvre dans différents contextes technologiques. Le choix d'un ADL et l'expression d'une architecture sont donc réduits à un contexte d'utilisation. **Une architecture ne doit pas être exprimée de manière dépendante d'une solution technologique de mise en œuvre.**

Dans les deux types de propositions, l'expression des architectures est en règle générale réduite à une forme syntaxique. Ceci constitue un frein à la collaboration des acteurs du processus logiciel dans le sens où il leur est difficile de travailler simultanément sur l'architecture. Cette forme syntaxique introduit une seconde difficulté : les différentes préoccupations architecturales ne sont pas structurées au sein de la description d'une architecture. Les contributions des différents acteurs se retrouvent mélangées au sein de cette description. Il est donc difficile d'avoir pour un acteur donné le point de vue associé sur l'architecture. La contribution d'un acteur n'est donc pas exprimable indépendamment des autres préoccupations. La difficulté associée à une activité est accrue. **L'aspect syntaxique et monolithique des langages de description d'architectures réduit les capacités de collaboration des acteurs du processus logiciel.**

1.5 Motivations et objectifs de notre proposition

1.5.1 Défis

Les motivations de ce travail visent à répondre aux limitations des propositions actuelles liées à la manipulation des architectures présentées dans la section précédente, c'est-à-dire de **faciliter la collaboration des acteurs du processus logiciel autour de la manipulation des architectures applicatives**. Pour cela, notre proposition repose sur quatre objectifs :

- utiliser une version réifiée des architectures pour faciliter leur manipulation ;
- offrir à chaque acteur du processus logiciel sa vision de l'architecture, *i.e.* présentant ses propres préoccupations ;
- permettre de fournir un environnement de manipulation des architectures adapté aux besoins d'un processus logiciel donné ;
- permettre l'expression d'architectures indépendamment de toute solution technologique.

Disposer d'une version réifiée de la représentation des architectures facilite leur manipulation. Mise à disposition dans un référentiel, cette version est plus facilement partageable entre les différents acteurs du processus logiciel. Cette approche apporte deux avantages. Premièrement, chaque acteur dispose ainsi de la version la plus à jour de l'architecture. Deuxièmement, chaque acteur peut contribuer de manière simultanée à la définition d'une architecture. Cette version est donc préférable à une version syntaxique pour le partage d'information entre les différents acteurs du processus logiciel. Enfin, un certain nombre d'outils peuvent interagir directement et simplement avec une version réifiée. De plus, ces outils peuvent partager une même représentation réifiée de l'architecture. **Fournir une version réifiée des architectures améliore la collaboration entre les différents acteurs.**

L'utilisation simultanée de la représentation d'une architecture par les différents acteurs pose des problèmes de concurrence quant à l'accès et à la modification d'informations partagées. Le découpage du processus logiciel a mis en avant que les préoccupations de chaque acteur sont distinctes 1.2. La structuration de la représentation des architectures en suivant la séparation de ces préoccupations apporte donc un avantage dans son utilisation. Les activités de chaque acteur ne rentrant pas en conflit, leur collaboration autour de l'architecture est facilitée. La mise en œuvre de la séparation des préoccupations serait difficile dans le cas d'une représentation non réifiée des architectures. **La séparation des préoccupations facilite la collaboration des acteurs.**

Afin d'être adapté au processus logiciel, l'environnement de manipulation des architectures doit être défini en fonction de ce processus. L'utilisation d'un environnement généraliste amène toujours des manques. Un support à la définition d'environnements adéquats est un élément important pour répondre exactement aux besoins d'un processus logiciel. Ce support doit permettre de mettre en œuvre la représentation réifiée la plus en phase avec les besoins du processus logiciel. La structuration de cette représentation en suivant la séparation des préoccupations doit respecter les préoccupations identifiées dans le processus logiciel. **Fournir un cadre de spécification et de production permet de disposer d'un environnement de manipulation des architectures adapté au processus logiciel.**

En dernier lieu, il nous semble important de définir les architectures indépendamment des solutions technologiques utilisées pour leur mise en œuvre. Ainsi, la pérénité des architectures

est accrue. Cette approche permet aussi de concevoir des architectures indépendamment des considérations techniques d'implantation des applications. Ce point tend à contribuer à la définition d'architectures de qualité, c'est-à-dire à donner la préférence aux concepts plutôt qu'aux détails. Une architecture n'est donc ni limitée ni restreinte au contexte du modèle technologique de mise en œuvre. **Offrir un environnement de définition des architectures indépendamment des technologies de mise en œuvre rend ces architectures plus pérennes.**

1.5.2 Propositions

Afin de relever ces défis, notre proposition repose sur la mise en œuvre conjointe de deux approches : l'utilisation des techniques de méta-modélisation et le respect de la séparation des préoccupations.

1. La base de notre proposition est la fourniture d'un cadre de définition d'environnements de manipulation des architectures logicielles. La définition de ces environnements repose sur les techniques de méta-modélisation. Un méta-modèle d'architectures décrit les concepts utilisables pour définir des architectures. Ces concepts correspondent aux besoins issus du processus logiciel et du domaine cible des applications. De par la définition des concepts requis, et uniquement de ceux-ci, la solution ainsi fournie est en adéquation avec les besoins.
2. La structuration d'un méta-modèle d'architectures est organisé ici sous la forme de plans. Cette structuration respecte la séparation des préoccupations. Chaque plan correspond à une préoccupation particulière. Il représente le point de vue de l'un des acteur du processus logiciel. Notre cadre de travail définit la manière dont ces plans doivent être définis pour, d'une part, isoler les préoccupations et, d'autre part, permettre leur intégration dans le but de fournir une version complète des architectures. Autrement dit, notre cadre de travail énonce un certain nombre de règles à respecter pour définir une séparation cohérente des préoccupations afin de garantir leur intégration. Ces règles définissent à la fois l'organisation des concepts au sein d'un plan et la manière dont les différents plans sont mis en relation.
3. Un méta-modèle d'architectures représente aussi la base de la production des environnements. La structuration par séparation des préoccupations se retrouve donc intégrée à l'environnement produit. La spécification relative aux moyens de méta-modélisation définie par l'*Object Management Group* propose un certain nombre de transformations pour automatiser la production d'environnements supportant la réification de modèles à partir de la définition d'un méta-modèle. Cette approche est ici mise en œuvre pour produire de façon automatisée les environnements de manipulation des architectures à partir de la définition de leurs méta-modèles.

1.6 Organisation de ce document

Ce document est composé de deux parties principales. La première dresse un état de l'art des technologies à base de composants et des langages de description d'architectures. Elle présente aussi les différentes formes que peut prendre la séparation des préoccupations et

un cadre d'utilisation des techniques de méta-modélisation. La seconde partie présente notre proposition : l'approche suivie, une illustration de sa mise en œuvre et son utilisation sur un exemple. Enfin, une dernière partie regroupe les conclusions, contributions et perspectives de ce travail. Une liste des publication produites pendant cette thèse, ainsi qu'une liste des acronymes sont disponibles en annexe de ce document.

Partie I : Etat de l'art

Le chapitre 2 « **Modèles de composants** » discute la contribution de l'approche à base de composants vis-à-vis de l'industrie du logiciel. Il propose une synthèse des bénéfices apportés par cette approche et en caractérise différents aspects. Sur la base d'un nombre de modèles choisis pour leur contribution, il dresse un panorama des moyens offerts par ces modèles aux acteurs du processus logiciel dans la production des applications. Pour terminer, un bilan des limitations des modèles actuels et de leur utilisation est présenté.

Le chapitre 3 « **Langages de description d'architectures** » (ADLs) présente les moyens actuellement disponibles pour définir et manipuler des architectures logicielles. En s'appuyant sur un ensemble d'ADLs significatifs, il présente les préoccupations architecturales liées au processus logiciel prises en compte dans ces langages et les moyens associés offerts aux acteurs du processus. Le bilan de ce chapitre regroupe, en plus des différentes préoccupations prises en compte, les limitations de l'approche à base d'ADLs pour manipuler des architectures logicielles.

Le chapitre 4 « **Méta-modélisation, la vision de l'*Object Management Group*** » discute des techniques de méta-modélisation, en s'appuyant sur les propositions de l'OMG. Il présente les objectifs de cette approche, ainsi que la structuration proposée par l'OMG en séparant les préoccupations des modèles. Un bilan des bénéfices et limitations actuelles de cette approche est dressé.

Partie II : Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures

Le chapitre 5 « **La proposition CODeX** » offre une vision d'ensemble de notre proposition. Il présente notre démarche concernant la définition et l'utilisation d'environnements de manipulation d'architectures dans le cadre du processus logiciel. Les bénéfices de l'utilisation conjointe des techniques de méta-modélisation et de la mise en œuvre de la séparation des préoccupations sont discutés.

Le chapitre 6 « **Méta-modélisation d'un environnement de manipulation d'architectures** » montre comment notre proposition est mise en œuvre afin de définir un environnement de manipulation d'architectures. Il présente en détails la structuration d'un tel environnement. L'utilisation particulière dans notre contexte des techniques de méta-modélisation fournies par l'OMG est discutée.

Le chapitre 7 « **Production d'un environnement de manipulation des architectures** » présente la mise en œuvre de l'environnement défini dans le chapitre 6. Il illustre les bénéfices d'une approche à base de méta-modélisation pour la production automatisée des environnements de manipulation d'architectures.

Le chapitre 8 « **Mise en œuvre dans le cadre du CCM** » illustre l'utilisation de notre approche dans le cadre du modèle de composants CORBA défini par l'OMG. Il présente la manière dont l'environnement produit dans les chapitres précédents peut être spécialisé et utilisé dans le cadre d'un modèle technologique de composants particulier.

Première partie

Etat de l'art

Chapitre 2

Modèles de composants

Les composants logiciels constituent aujourd'hui une technologie phare de l'industrie du logiciel. Une raison de cette situation est la volonté de beaucoup d'industriels de capitaliser leur code et de fournir du logiciel sur l'étagère (*COTS, Commercial Off The Shelf*) pour accroître leur productivité et réduire le temps entre l'identification d'un besoin et la fourniture d'une solution.

Ce chapitre présente une vision microscopique du processus logiciel. Il présente les briques élémentaires des applications, les composants logiciels. Les composants logiciels représentent un bon moyen pour structurer les applications et améliorer leur production. Nous allons discuter dans ce chapitre des réponses apportées par les composants logiciels aux préoccupations d'un processus logiciel.

La définition du terme *composant logiciel* varie d'un contexte à un autre. Ce chapitre présente dans la section 2.1 une caractérisation des composants logiciels au travers d'un méta-modèle regroupant les différents aspects d'un composant logiciel. Chaque modèle ayant des préoccupations différentes, ce chapitre présente un certain nombre de modèles de composants. Ces modèles ont été choisis pour les préoccupations qu'ils adressent et regroupés en trois catégories.

- La section 2.2 présente les modèles académiques Darwin et JavaPod.
- La section 2.3 présente les modèles industriels JavaBean et Enterprise Java Beans (EJB) de Sun Microsystems ainsi que (Distributed) Component Object Model ((D)COM) de Microsoft.
- La section 2.4 présente les modèles de référence Open Distributed Processing de l'ISO et le CORBA Component Model (CCM) de l'Object Management Group.

Pour terminer, la section 2.5 de ce chapitre dresse un bilan de l'approche à base de composants logiciels. Il résume les différents concepts fournis par les modèles de composants et met en relation ces concepts et les modèles les supportant. Il résume aussi les préoccupations du processus logiciel prises en compte par les modèles de composants.

2.1 Un modèle abstrait de composants

2.1.1 Caractérisation des composants

Comme nous l'avons déjà mentionné dans l'introduction, il n'existe pas, aujourd'hui, *une* définition du terme « composant logiciel ». Nous reprenons donc comme base les deux

définitions suivantes.

> *Un composant est un morceau de logiciel assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour que l'on puisse l'installer et en assurer le support. De plus, il est doté d'interfaces standards pour pouvoir interopérer.* (Jed Harris, président du CI Lab, 1995)

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* (*Workshop on Component Oriented Programing* [91], 1996)

Ces deux définitions ne sont pas très précises quant à ce que l'on peut qualifier de composants logiciel. Il en résulte que les composants vont de simples morceaux de programmes sous forme binaire (Microsoft) à des entités logicielles indépendantes et coopérantes ayant une existance propre au sein d'un système (Object Management Group). Le terme « composant logiciel » est de plus utilisé depuis la phase de conception jusqu'à la phase d'exécution d'une application, il prend donc plusieurs formes.

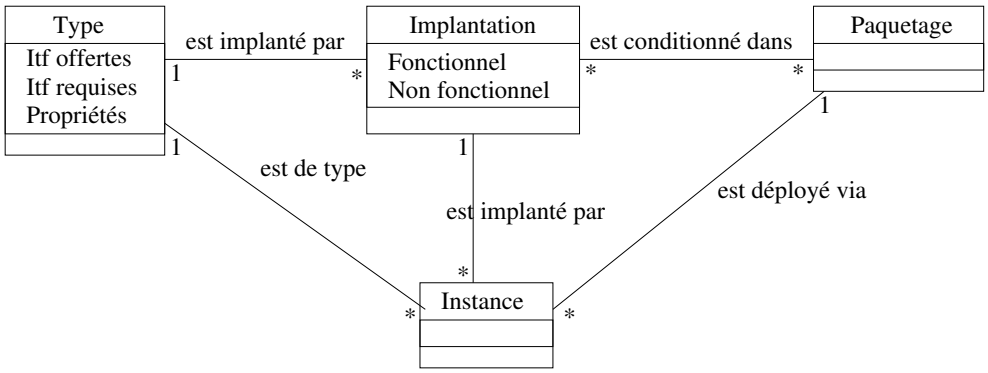


FIG. 2.1 – *Méta-modèle d'une caractérisation des composants logiciels*

Quatre notions relatives à un composant logiciel étendent les deux définition précédentes : type de composant, implantation de composant, paquetage de composant et instance de composant. La figure 2.1 présente un méta-modèle de cette caractérisation. Les différents éléments de ce méta-modèle sont discutés tout au long de cette section. Ce méta-modèle précise qu'un type de composant est réalisé par une ou plusieurs implantations, chacune d'entre elles ne réalisant qu'un type de composant. Chaque implantation est conditionnée dans un ou plusieurs paquetages et un paquetage peut contenir plusieurs implantations. Enfin, une instance de composant est déployée au travers d'un unique paquetage (qui peut servir à déployer plusieurs instances) et implantée par une unique implantation de composant (qui peut être utilisé pour créer plusieurs instances). Implicitement, du fait de l'unicité de son implantation, une instance est d'un unique type de composant (qui peut être instancié plusieurs fois).

2.1.1.1 Type de composant

Un *type de composant* est la définition abstraite d'une entité logicielle. Il est caractérisé par trois éléments : ses interfaces, les modes de coopération avec les autres types de composants et ses propriétés configurables. Il est important qu'une interface soit définie de manière indépendante à toute implantation. D'une part, il est ainsi possible de fournir plusieurs implantations pour un même type de composant et, d'autre part, une implantation de composant peut être substituée par une autre implantation d'un type compatible. Les interfaces d'un type de composant sont idéalement de deux ordres, indispensables pour réaliser des assemblages :

- les interfaces fournies par le composant sont du même ordre que les interfaces des objets : elles définissent les services fournis par le type de composant, en énumérant les signatures des opérations, ainsi que les différentes données entrantes et/ou sortantes du composant pour chaque opération ;
- les interfaces requises par le type de composant représentent un progrès par rapport à l'approche objet. Dans le cas des objets, une référence sur un objet utilisé est enfouie au cœur du code. Dans le cas des composants, les interfaces des types de composants utilisés devraient être exprimées au niveau du type de composant. Il est ainsi plus aisé, d'une part, de substituer un composant par un autre et, d'autre part, de gérer les connexions entre des types de composants, *i.e.* de connaître les dépendances pour l'installation et le remplacement, ce qui représente un aspect important pour l'administrateur.

Chaque interface définit un mode de communication avec les autres types de composants. Dans notre contexte, trois modes de communication sont souhaitables au minimum : le mode synchrone (par exemple l'invocation de méthode), le mode asynchrone (par exemple l'émission d'événements) et le mode diffusion en continu (par exemple les flots de données). Aucun des modèles discutés par la suite ne propose d'autre mode de coopération comme la synchronisation ou le partage d'informations au travers d'un tableau noir (très utilisé dans le paradigme agent par exemple). Ces deux modes nous semblent plus relever de services systèmes dans un contexte composant. Ainsi, ces modes de coopération ne sont pas définis ni mis en avant au niveau des modèles de composants (ce qui ne représente pas nécessairement le meilleur choix de mise en œuvre puisqu'une définition au niveau du modèle simplifierait peut-être leur utilisation).

Le dernier aspect caractérisant un type de composant est l'ensemble de ses propriétés configurables. Ces propriétés vont, en partie, paramétrer le comportement d'une instance de composant dans un contexte donné. Afin de ne pas modifier l'implantation d'un type de composant pour chaque contexte, la phase de configuration de ces propriétés permet donc d'adapter l'implantation d'un composant à son contexte et de préciser le comportement des instances. La configuration d'une instance de composant repose essentiellement sur le fait de fixer les valeurs initiales des propriétés (positionnement d'attributs au sens des objets ou bien invocation de traitements pour fixer un état initial). La phase de configuration inclut aussi la fourniture des services systèmes requis par l'instance de composant (recherche des références et initialisation de l'utilisation de ces services). Cette phase est indépendante de l'instanciation dans le sens où la création d'une instance est une action générique alors que la configuration peut requérir la fourniture de paramètres spécifiques, et donc variables, selon les instances.

2.1.1.2 Implantation de composant

L'implantation d'un type de composant regroupe deux notions : l'*implantation fonctionnelle* et l'*implantation non fonctionnelle*. L'implantation fonctionnelle d'un composant représente sa mise en œuvre d'un point de vue métier, indépendamment des conditions d'exécution. Elle représente l'ensemble des interfaces fournies et requise, ainsi que les propriétés d'un type de composant. Les traitements de l'implantation fonctionnelle sont en règle générale programmés. L'implantation non-fonctionnelle représente l'adaptation de ce code métier aux conditions d'exécution [26]. Les traitements liés à cette seconde partie de l'implantation d'un composant sont idéalement décrits et non programmés.

Des propriétés comme la persistance, les besoins transactionnels ou la sécurité liés à un composant peuvent être décrits et leur prise en charge automatisée. La prise en charge automatisée des propriétés non fonctionnelles d'un composant peut s'appuyer sur la spécification du type de composant, mais doit aussi prendre en compte les besoins de l'implantation fonctionnelle. Il est donc nécessaire de décrire les besoins de l'implantation de composant, et de fournir un canevas pour l'intégration symbiotique de l'implantation non fonctionnelle et de l'implantation fonctionnelle.

Considérons l'exemple d'un porte-monnaie électronique ; l'implantation fonctionnelle regroupe la réalisation des services tels le débit ou le crédit d'une somme. L'implantation non fonctionnelle regroupe quant à elle les aspects relatifs à la qualité de service comme les besoins transactionnels, de sécurité et de persistance. La gestion de la communication (aspect non fonctionnel) entre un client et le porte monnaie est automatisable à partir de l'interface du porte-monnaie. La gestion de l'aspect de persistence provient de la description de l'état abstrait du porte-monnaie. La prise en charge de la sécurité et des transactions provient aussi de la description du porte-monnaie : les opérations crédit et débit doivent être sécurisées (contrôle d'accès) et exécutés dans un contexte transactionnel.

2.1.1.3 Paquetage de composant

Un *paquetage de composant* est une entité diffusable et déployable, bien souvent une archive, contenant la définition du type de composant, au moins une implantation de ce type et une description du contenu du paquetage. La description regroupe le type de composant et les contraintes techniques de l'implantation. Ces contraintes peuvent exprimer que le composant est implanté en C++, pour le système SUN Solaris et nécessite la bibliothèque dynamique *libFoo.so.1*.

L'utilisation de paquetages logiciels permet de rendre diffusable un composant dans sa forme exécutable. Ici encore, un gain apparaît par rapport aux objets : un composant est diffusable de manière unitaire et sous forme exécutable. Il n'est pas nécessaire de connaître l'implantation du composant pour l'intégrer dans une application ; seule la connaissance de son type et éventuellement de ses contraintes est requise.

2.1.1.4 Instance de composant

Une *instance de composant* est, au même titre qu'une instance d'objet, une entité existante et s'exécutant dans un système. Elle est caractérisée par une référence unique, un type de composant et une implantation particulière de ce type. De même que les objets, une instance

de composant peut recevoir des invocations d'opérations. Une différence importante entre instance de composant et d'objet est le fait qu'un modèle de composants se doit de rendre le cycle de vie d'une instance moins opaque. Le cycle de vie d'une instance de composant est idéalement descriptible et configurable en fonction de son contexte d'utilisation. L'utilisation de patrons de conception [29], tels que les patrons *fabrique* et *recherche*, représente une réponse très intéressante au support du cycle de vie.

2.2 Modèles de composants académiques

Les modèles de composants académiques représentent des expérimentations sur des aspects précis dans le but de mieux comprendre les composants et de contribuer par la suite à la définition de modèles plus généraux comme les modèles industriels.

2.2.1 Darwin

Conic puis Darwin (1990) [42] sont des précurseurs des systèmes de configuration à base de composants logiciels. Ils ont été développés au Distributed Software Engineering Group de l'Imperial College à Londres, Grande-Bretagne. Darwin propose un des premiers modèles de composants pour la construction d'applications réparties.

2.2.1.1 Modèle

Dans le contexte de Darwin, le composant représente avant tout une unité de traitement. Un composant est donc assimilé à un processus (au sens système du terme). Un composant est décrit par une interface qui contient les services fournis et requis, s'apparentant aux entrées et sorties de flots de communication. Deux types de composants existent : les *primitifs* qui intègrent du code logiciel, et les *composites* qui sont des interconnexions de composants primitifs ou composites.

Un composant primitif encapsule le code applicatif de la fonction qui lui a été attribuée ; un composant composite encapsule des instances de composants composés ou primitifs. Pour chacun de ces composants, une interface identifiant les services fournis et requis par ce composant est définie. Les composants composites sont des unités de configuration ; ils contiennent la description des composants d'une application ainsi que leurs interactions. Une application est donc un composant composite. Ces composants coopèrent selon un modèle de communication connu par la signature des composants.

Une caractéristique importante de Darwin est la capacité de décrire la création dynamique de composants par l'application. Cette caractéristique apporte une amélioration notable puisque l'architecture d'une application n'est plus considérée comme un ensemble de composants prévus dès la phase de conception. L'instanciation dynamique est une pré-déclaration des instances qui seront effectivement créées non pas lors de l'initialisation du composite, mais lors du premier appel à ces composants.

2.2.1.2 Mise en œuvre du modèle

Darwin décrit un langage de configuration à part entière. La définition d'un type composant suit la syntaxe de la figure 2.2.

```
Component nom (liste de parametres) {
    provide nomPort <port, signature> ;
    require nomPort <port, signature> ;

    // implementation : vide dans le cas d'un primitif,
    // composée de déclarations d'instances et de schéma
    // d'interconnexion dans le cas de composites.
}
```

FIG. 2.2 – *Description d'un composant avec Darwin*

Les services requis ou fournis correspondent à ceux que le composant nécessite pour fonctionner, ou offre à d'autres composants. Les services n'ont pas de conotation fonctionnelle, et désignent seulement le type de communication utilisé ou autorisé à venir appeler une opération du composant (granularité des ports). Ces communications sont fournies par le support d'exécution qui permet à des configurations Darwin de s'exécuter.

Les composants primitifs (classes C++) héritent d'une classe « Process ». A l'intérieur du code des composants, il est fait directement référence aux ports de communication requis ou fournis. La sémantique associée à un composant primitif est le processus. L'instanciation d'un composant correspond à la création d'un nouveau processus. Le composant primitif encapsule du code applicatif.

L'exemple suivant est directement tiré de [47]. Le composant de base (primitif) est un filtre dont l'interface est composée d'un port d'entrée `left` qui récupère un entier. Si cet entier répond au critère du filtre composant alors le composant le transmet par le port de sortie `output`, sinon il est transmis via le port `right`. Le composant (composite) `Pipeline` est une composition d'un nombre paramétrable de filtres. La syntaxe de ce composant est décrite dans la figure 2.3. Ce composant `Pipeline` a pour rôle de chaîner des filtres les uns avec les autres.

Enfin, la figure 2.4 décrit cet exemple sous forme graphique. Chaque boîte représente un composant. Les ronds blancs et noirs représentent respectivement les services requis et fournis. Ces services sont reliés afin de montrer leur coopération.

2.2.1.3 Evaluation

En plus d'être un des précurseurs des modèles de composants, Darwin propose trois contributions majeures :

- l'utilisation de la **notion de port** pour exprimer les points de connexion des composants ;
- la définition explicite des **services requis** par un composant (mot clé `require`) ;
- la mise en œuvre du **concept de composite** pour structurer les applications.

```
Component pipeline (int n) {
    // Interface
    provide input ;
    require output ;
    // implementation
    array F[n] : filter ; // ens. d'instances de filtre
    forall k: 0..n-1 {
        inst F[k]; // création d'une instance de filtre
        // lien avec le composant pipeline
        bind F[k].output - output ;
        when k<n-1
            bind F[k].right - F[k+1].left ;
    }
    bind input - F[0].left ;
    bind F[n-1].right - output ;
}
```

FIG. 2.3 – *Définition d'un composant composite avec Darwin*

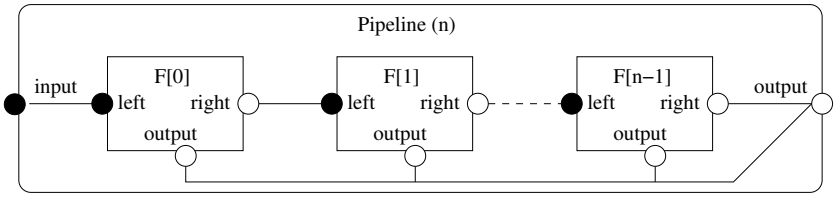


FIG. 2.4 – *Représentation graphique avec Darwin*

Parmi les points négatifs, Darwin propose un modèle de composants à grosse granularité, un composant représente un processus. Ces composants ne sont pas packagés pour faciliter le processus de déploiement. Ce dernier n'est pas pris en charge par Darwin. Ensuite, l'intégration de code existant est relativement difficile. Le code permettant l'utilisation de l'environnement Darwin n'est pas transparent pour le programmeur. Celui-ci doit explicitement utiliser les objets de type « port » pour réaliser ses communications et effectuer les créations de composants lorsqu'il les sollicite. De plus, le langage de programmation des composants doit être le même que celui qui définit les classes et objets manipulés par l'environnement, en l'occurrence C++. Enfin, l'utilisation de divers modes de communication n'est pas configurable. Il faut que le support d'exécution supporte différents types de port.

2.2.2 JavaPod

Entre 1998 et 2001, le projet SIRAC de l'INRIA Rhône-Alpes, à travers le projet JavaPod [12, 11], s'est intéressé aux aspects fonctionnels et non fonctionnels des approches à base de composants, séparation qui commençait à apparaître. Ce projet propose une plate-forme dont l'architecture est inspirée de l'architecture EJB (voir section 2.3.2) et qui est mise en oeuvre grâce à un modèle original de composition d'objets implantés par une extension de Java. L'objectif de ce modèle est de pouvoir offrir aux applications un ensemble non limité *a priori* de propriétés non fonctionnelles, et également de pouvoir composer facilement ces différentes propriétés.

2.2.2.1 Modèle

La définition des composants dans le projet JavaPod s'appuie sur le modèle ODP (voir la section 2.4.1) ; il désigne un objet pouvant avoir plusieurs interfaces d'accès. Un *composant* peut être constitué d'un ou plusieurs objets, mais doit rester non distribué. Les composants sont reliés par des connecteurs, qui sont, eux, des objets distribués.

Ce projet propose un modèle de composition afin de pouvoir offrir aux applications, par assemblage d'extensions, un ensemble non limité *a priori* de propriétés non fonctionnelles. L'ensemble de ces propriété est, de plus, modifiables dynamiquement. Ce modèle permet de composer des objets, au sens général du terme (c'est-à-dire une entité), qui encapsulent un état interne, accessible uniquement par des méthodes. Le résultat de la composition d'un certain nombre d'objets est appelé un *objet composite*. Les objets internes d'un objet composite sont appelés ses *constituants*. Ces constituants sont totalement ordonnés. Le terme d'*objet extensible* désigne le constituant le plus « petit » pour cet ordre. Les autres constituants sont appelés des *extensions*. L'objet extensible est invariable, alors que les extensions peuvent être ajoutées, supprimées ou remplacées dynamiquement. La figure 2.5 représente les constituants les uns au dessus des autres, l'objet extensible étant à la base.

Un objet composite est également un objet : ses méthodes sont l'union des méthodes de ses constituants, et son état interne est l'union des états internes de ses constituants. La sémantique d'un appel de méthode sur un objet composite est la suivante :

- si la méthode appelée n'est définie dans aucun constituant, l'appel échoue ;
- sinon, l'appel est exécuté par le constituant le plus grand selon l'ordre qui définit cette méthode.

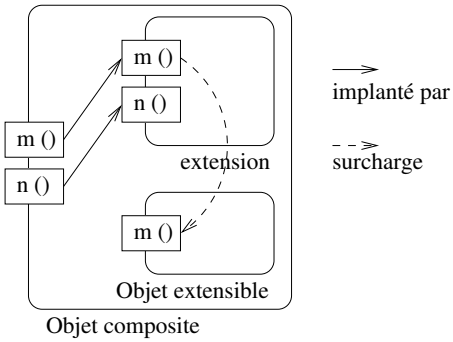


FIG. 2.5 – *Modèle d'un composant JavaPod*

Autrement dit, si une méthode est définie dans plusieurs constituants, le constituant le plus haut masque, ou surcharge, les définitions des autres constituants. Cependant il est possible, dans le cas d'une extension, d'appeler la méthode surchargée par un appel spécial. La sémantique d'un tel appel est alors le même que précédemment, mais en se limitant aux constituants situés en dessous de l'extension appelante. On considère que l'état interne d'un constituant n'est accessible que par ses méthodes.

2.2.2.2 Mise en œuvre du modèle

L'architecture des JavaPod, représentée dans la figure 2.6, est constituée d'un certain nombre d'éléments qui sont la mise en oeuvre, au niveau de la plate-forme logicielle, des concepts du modèle de programmation. La figure 2.6 présente ces différents éléments. Comme le montre cette figure, une plate-forme à composants est constituée de quatre types d'éléments, qui ont une existence concrète à l'exécution : des *serveurs*, des *conteneurs*, ainsi que des *talons* et des *squelettes* qui, regroupés, forment les connecteurs.

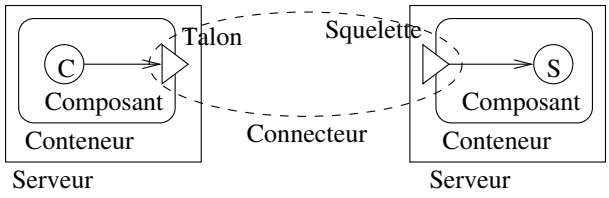


FIG. 2.6 – *Architecture de la plate-forme JavaPod*

Le serveur est une structure d'accueil, un support d'exécution pour les conteneurs, qui sont eux-mêmes une structure d'accueil pour composants. Un serveur fournit tous les services systèmes dont peuvent avoir besoin les conteneurs : protocoles de communication, service de persistance, gestion de ressources, *etc.*

Un conteneur est la partie système correspondant à un composant. Le conteneur encapsule le composant, au sens où toutes les interactions du composant avec l'extérieur doivent normalement passer par le conteneur. Grâce à cette interposition, le conteneur peut gérer les

propriétés non-fonctionnelles du composant : modèle d'exécution, de persistance, de synchronisation, *etc.*

Un talon ou un squelette est à l'interface entre un conteneur et un connecteur, et appartient aux deux à la fois. Un talon permet à un composant d'envoyer des messages ou des données vers l'extérieur, alors qu'un squelette permet d'en recevoir. Chaque talon ou squelette a un type, qui est un ensemble de signatures de méthodes. Ce type définit les méthodes que le composant peut appeler sur un talon, ou qu'un squelette peut appeler dans un composant. Le modèle de composition a été intégré dans un nouveau langage, appelé « ejava » car c'est une extension de Java. L'architecture de la plate-forme à composants JavaPod a été implantée, en ejava, sous forme d'un noyau minimal fournissant une classe d'objet extensible par élément de l'architecture. Le langage ejava est conçu pour pouvoir définir des objets composites, il s'agit d'un sur-ensemble de Java.

2.2.2.3 Evaluation

Ce modèle récent s'attache à offrir une solution permettant de développer des composants logiciels sans tenir compte des parties non fonctionnelles de ceux-ci. Il permet la conception et l'implantation des composants ainsi que leur assemblage. Un avantage significatif est la **non limitation des aspects non fonctionnels** intégrables, à la différence du modèle EJB (cf. section 2.3.2) et CCM (cf. section 2.4.2). Pour cela, ce modèle propose de construire chaque élément de l'architecture par composition d'extensions.

Le second aspect important de ce projet est la **réification des connecteurs** à l'exécution. Les connecteurs existent au même titre que les composants et sont manipulables. A l'exécution, les propriétés des connecteurs peuvent être modifiées.

Le modèle de composant, dans le cadre de cette plate-forme, est plus ouvert et plus flexible que l'héritage et la délégation. La proposition consiste à étendre le langage Java pour la définition du modèle de composition. Le déploiement des composants JavaPod est décrit par des scripts précisant les assemblages, composants et connecteurs, à mettre en œuvre. Pour les autres étapes du processus logiciel, JavaPod prend en charge les différents aspects au même titre que le langage Java.

2.3 Modèles de composants industriels

Les modèles de composants industriels sont une réponse aux besoins du marché. Ils sont donc certainement moins novateurs que les modèles académiques mais ont pour principale motivation de permettre la production d'applications réelles pouvant être utilisées dans l'industrie.

2.3.1 JavaBeans

Le modèle des JavaBeans [36] est un modèle de composants proposé par Sun Microsystems dès 1996.

2.3.1.1 Modèle

La définition d'un JavaBean donnée par Sun est la suivante :

> *A JavaBean is a reusable software component that can be manipulated visually in a builder tool.*

Un Java Bean possède une interface. Il peut disposer de méthodes pour offrir des traitements, et d'attributs pour représenter ses propriétés (voir figure 2.7). La coopération entre instances repose sur l'utilisation d'événements. Tout comme en Java standard, les événements sont mis en oeuvre de manière synchrone.

Au niveau de l'interface, rien ne différencie un Bean de tout autre objet Java. Toute personne intéressée par les fonctionnalités d'un Bean, et ne connaissant pas les JavaBeans, pourra sans difficulté utiliser ce Bean selon le modèle objet traditionnel de Java. L'utilisation de l'introspection sur un Bean permet de mieux souligner ses capacités : mise en évidence des propriétés, expression des ports, *etc.* Cette introspection se fait au travers de l'interface BeanInfo.

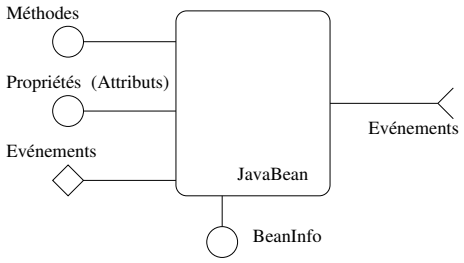


FIG. 2.7 – *Modèle de composant JavaBean*

2.3.1.2 Mise en œuvre du modèle

La définition d'un JavaBean se fait de la même manière que la définition d'un objet Java. Les règles supplémentaires sont des patrons de conception, principalement syntaxiques. Les attributs sont définis par la présence d'une ou deux méthodes permettant leur lecture et éventuellement leur modification (patron de conception *accesseur*). De la même manière, une source d'événements est définie par la présence de deux opérations d'abonnement et de désabonnement aux événements. Un Bean consommateur d'événements devra implanter une interface de consommateur d'événements (`java.util.EventListener` ou une sous-interface). (Patron de conception observateur - observable.)

La spécification n'impose pas de définir des interfaces pour les Beans : il est possible de fournir uniquement leurs implantations. La figure 2.8 présente cependant une interface illustrant la définition d'un Bean.

L'environnement d'exécution des JavaBeans est simplifié au maximum : il lui suffit de disposer d'une machine virtuelle Java. Dans le cas d'un composant graphique, un JavaBean a besoin d'un conteneur graphique, par exemple une fenêtre ou un navigateur Web.

```
interface MonBean {
    // méthode pour lire l'attribut
    String getCouleur () ;

    // gestion des événements
    void addMonBeanListener (MonBeanListener l) ;
    void removeMonBeanListener (MonBeanListener l) ;
}
```

FIG. 2.8 – *Définition d'un JavaBean*

2.3.1.3 Evaluation

Les JavaBeans contribuent sur deux points au monde des composants logiciels :

- le mode de communication avec un Bean est précisé dans son interface, il peut être **événementiel** ;
- la définition de Beans repose sur l'utilisation de patrons de conceptions (*design patterns*) comme le patron **Observateur - Observable**.

Les Beans sont, de par leur définition, destinés à être assemblés de manière graphique pour produire des Beans plus complexes ou des applications (répondant bien aux besoins de la partie graphique cliente). Pour cela, un architecte exploite les méta-informations contenues dans les BeanInfo sur les services offerts, la gestion des événements par le Bean. Les étapes de conception, d'implantation des composants et de définition de l'architecture trouvent des réponses dans les Java Beans (interfaces Java et *Design Patterns*).

La fourniture d'un Bean se fait sous la forme d'un paquetage regroupant l'implantation du Bean, son interface si elle existe et des informations sur le Bean contenues dans une classe de type BeanInfo. Ce paquetage s'exprime sous la forme d'une archive Java (.jar) qui peut être distribuée, vendue et déployée. L'étape de déploiement des JavaBeans se fait en deux temps. La première, manuelle, revient à rendre disponibles des archives de beans sur un serveur Web ou un système de fichiers. La seconde, automatique, correspond au chargement des beans dans un browser lors de l'accès à la page Web correspondante ou dans une machine virtuelle Java (JVM, *Java Virtual Machine*). La répartition des applications tient essentiellement dans le fait qu'une partie cliente de l'application composée de Beans a besoin de dialoguer avec la partie serveur.

Avant de vouloir fournir un modèle, Sun propose avec les JavaBeans une méthodologie d'utilisation de son langage et environnement phare : Java. Les JavaBeans répondent bien au problème de la production des parties clientes des applications, et fournissent une approche assez simple pour développer des interfaces graphiques à base de composants potentiellement réutilisables. Cependant, les JavaBeans représentent une solution propriétaire qui ne facilite pas nécessairement l'intégration de l'existant. Au même titre que le choix de COM (voir section 2.3.3) est guidé par une plate-forme, le choix des JavaBeans est guidé par un langage.

2.3.2 Enterprise Java Beans

De par leur définition, les JavaBeans représentent un modèle de composant client. Les JavaBeans n'étant pas en adéquation avec la réalisation de composants serveurs, Sun a mis au point les Enterprise Java Beans (EJB) [55] dont la spécification 1.0 est sortie en novembre 1997. Ce modèle représente un canevas de composants serveurs aussi appelés composants métier pour concevoir des applications distribuées, orientées transactions et base de données.

2.3.2.1 Modèle

Contrairement aux JavaBeans, un EJB est nécessairement représenté par une interface Java : l'interface distante du Bean qui définit la vue cliente de l'EJB. Cette interface hérite de plusieurs interfaces soit prédéfinies, comme *EJBObject*, soit définies par l'utilisateur, interface regroupant les opérations métier d'un EJB. La fourniture d'opérations métier est l'objectif de la conception d'un EJB. L'implantation de ces opérations représente l'implantation du composant.

En plus de cette interface métier, un EJB offre une interface pour accéder aux méta-données de l'instance de composant, et une interface qui gère le cycle de vie d'un composant (*maison* ou *home interface*). L'interface de maison permet de créer (ou de rechercher dans le cas d'un composant persistant) et de détruire une instance de composant. Elle offre aussi une opération retournant une référence sur l'interface de méta-données. Cette dernière interface permettra par la suite de construire dynamiquement des requêtes sur le composant.

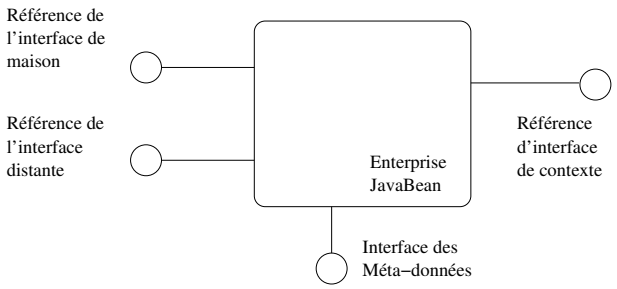


FIG. 2.9 – *Modèle abstrait des Enterprise Java Beans*

Les implantations composants sont de deux types : composants de session et composants entités. Les instances des premiers sont créées à chaque connexion d'un client. Elles peuvent être avec ou sans état (composant de traitement). La terminaison d'une session prend place lorsque le client invoque la destruction de l'instance de composant. Les instances du second type disposent d'un identifiant unique permettant au client de retrouver une instance particulière. Le cycle de vie de ces dernières est géré par le conteneur.

Non seulement les EJBs fournissent un canevas de conception des composants métier, mais ils favorisent aussi la conception d'implantations de composants réutilisables. En effet, la spécification 1.1 souligne l'importance de pouvoir modifier le comportement d'un EJB sans modifier le code métier de celui-ci. Pour cela, l'interface de méta-données permet de fixer le comportement d'un EJB au déploiement.

Le second moyen d'agir sur le comportement d'un EJB est de modifier ses propriétés non fonctionnelles. Dans cette optique, les politiques de sécurité et la démarcation des transactions peuvent être fixées au sein de son descripteur.

2.3.2.2 Mise en œuvre du modèle

La déclaration de l'interface distante d'un EJB se fait au travers d'une interface Java qui étend l'interface `javax.ejb.EJBObject`. La déclaration d'une maison de composant se fait de la même manière au travers d'une interface Java qui étend `javax.ejb.EJBHome`. Ces déclarations sont présentées dans la figure 2.10.

```
// Interface distante de l'EJB RepertoireBean
public interface RepertoireBean
    extends javax.ejb.EJBObject {

    public void ajouterNom (String nom)
        throws RemoteException ;
    public String[] listeNoms ()
        throws RemoteException ;
}

// Interface de maison de l'EJB RepertoireBean
public interface RepertoireBeanHome
    extends javax.ejb.EJBHome {

    RepertoireBean create ()
        throws RemoteException, CreateException ;
}
```

FIG. 2.10 – *Définition d'un Enterprise Java Bean et de sa maison*

L'environnement d'exécution se découpe en deux parties : un ou plusieurs conteneurs (structure d'accueil) et un serveur (type serveur d'application). Les différents conteneurs sont présents dans le serveur. Plusieurs serveurs peuvent être mis en oeuvre pour une application, mais en général il semble qu'un seul serveur par hôte soit le bon choix (un serveur peut être *multi-threadé*).

Un utilisateur ne dialogue pas directement avec une instance d'EJB. Le client utilise une interface distante (*remote interface*). Cette interface délègue les requêtes à l'instance d'EJB. Cette délégation est prise en charge par le conteneur qui a un rôle de médiateur. Ce rôle correspond à la prise en charge des propriétés non fonctionnelles d'un EJB, comme la persistance.

Un conteneur peut recevoir plusieurs types d'EJBs. Comme le montre la figure 2.11, l'architecture de l'environnement d'exécution est en couches : les EJBs, les conteneurs et les serveurs. Les conteneurs interviennent dans les échanges entre client et instance d'EJB alors que les serveurs sont uniquement mis en oeuvre pour héberger les conteneurs. Les conteneurs sont localisés par l'application via l'API (Application Programming Interface) *Java Naming and Directory Interface* (JNDI) pour pouvoir être utilisés par la suite. Un EJB présente deux

interfaces à ses clients : une interface de maison et une interface distante offrant accès à ses traitements.

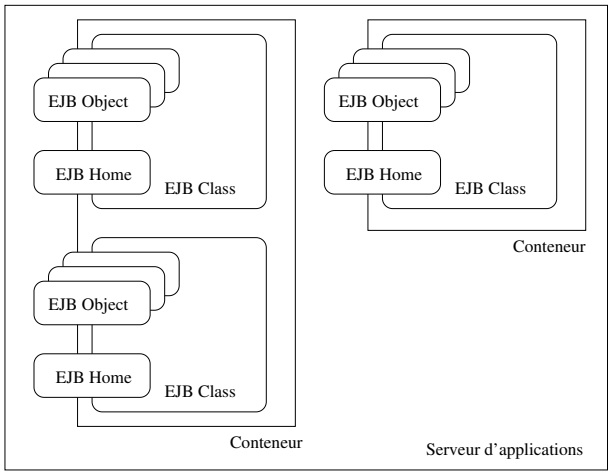


FIG. 2.11 – *Environnement d'exécution des Enterprise Java Beans*

Dans le but de faciliter leur déploiement, les Beans sont décrits dans un descripteur qui regroupe d'une part des informations générales sur le fournisseur du Bean, mais aussi une description des besoins du Bean en termes de ressources : démarcation des transactions, politiques de sécurité, ressources systèmes. La spécification précise qu'un Bean ne doit jamais accéder directement aux ressources système, mais à des objets encapsulant ces ressources. Pour utiliser ces ressources, les instances de Beans s'adressent à des fabriques de ressources pendant la phase d'exécution.

La recherche des ressources de l'environnement par un Bean est programmée au sein de son implantation. Le descripteur exprime les ressources nécessaires à un Bean pour s'exécuter, qui doivent être présentent lors de l'installation d'un Bean. La mise à disposition de ces ressources est à la charge de l'installateur de Bean. Lors de l'exécution d'un Bean, celui-ci va rechercher et utiliser ces ressources.

2.3.2.3 Evaluation

Le modèle EJB représente un grand pas vers un modèle opérationnel de composants serveurs. L'utilisation d'environnements d'exécution plus ou moins génériques pour accueillir des composants, la prise en compte du déploiement, ainsi que la **gestion de certaines propriétés non fonctionnelles** sont autant d'atouts pour produire des composants réutilisables. De plus, les EJBs ne fournissent pas seulement un **canevas pour concevoir des composants métier**, mais favorisent aussi la conception d'implantations de composants réutilisables. Un même code métier peut être utilisé dans différents contextes sans modification, ses propriétés non fonctionnelles étant fixées au travers de son descripteur.

Cependant, les EJB restent une solution propriétaire de Sun, uniquement destinée à une utilisation en Java. Le faible nombre de propriétés non fonctionnelles reflète certainement une demande du marché, mais qu'en est-il de l'ajout de nouvelles propriétés dans le modèle ? Un

second regret quant à la mise en œuvre est l'enfouissement de la connectique au sein même des implantations. L'expression des besoins par description résoud en partie le problème de la disponibilité des besoins à l'exécution, mais il ne semble pas souhaitable qu'un composant gère sa connectique, auquel cas celle-ci se retrouve figée. Enfin, le déploiement d'applications à base d'EJBs paraît assez statique, et la mise à disposition des paquetages sur leur site d'exécution reste une opération manuelle. Globalement, les EJBs fournissent des réponses, même si elles sont parfois basiques comme les interfaces Java ou le mécanisme de déploiement, pour les différentes étapes du processus.

2.3.3 (D)COM/COM+

COM (Component Object Model) [85] et DCOM (Distributed COM) [34] représentaient la proposition de Microsoft en termes de composants logiciels. La proposition COM résulte d'une évolution progressive de l'environnement MS-Windows vers les composants. Les deux sources de COM sont d'une part la composition de contrôles (non orientés objet) Visual Basic, et l'utilisation de documents composites (OLE-Object Linking and Embedding). Toutes ces propositions se retrouvent désormais plus ou moins dans la nouvelle proposition .Net [97] que nous ne discutons pas ici.

2.3.3.1 Modèle

Du fait de son processus de conception, COM ne s'appuie pas sur un réel modèle. Dans le cadre de COM, un composant est essentiellement une entité binaire pour laquelle sont définis plusieurs interfaces et un mode d'interaction. Ces deux éléments se résument en fait par un simple pointeur, offrant accès aux fonctions de la bibliothèque, implantation du composant. COM n'impose pas de contrainte sur l'implantation du composant : le composant peut être implanté sous la forme d'une ou plusieurs classes, sous la forme d'une bibliothèque de procédures ou de fonctions, *etc.* Un composant peut, par exemple, avoir plusieurs interfaces. Une interface principale est nécessairement disponible sur un composant, `IUnknown`. Cette interface offre une opération `QueryInterface` qui permet de découvrir les interfaces fournies, et ainsi, de naviguer entre elles.

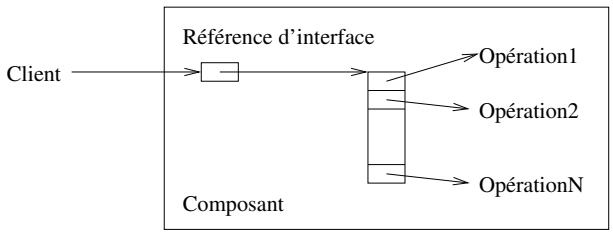


FIG. 2.12 – *Modèle binaire des composants COM*

Dans le but d'une utilisation générique, un composant peut fournir l'interface `IDispatch`. Cette interface regroupe l'utilisation de toutes les opérations d'un composant sous une unique opération : `invoke()`. Cette opération permet l'utilisation systématique et dynamique d'un composant à partir d'un langage interprété comme Visual Basic. L'utilisation de cette méthode

permet d'autre part de faire du forward automatique d'opérations sur un composant : il n'est pas nécessaire de connaître statiquement son interface pour l'utiliser.

De manière optionnelle, un composant peut déclarer des « outgoing interfaces ». Ces interfaces, sont des interfaces utilisables par le composant. Elles sont exploitées par le composant s'il est connecté à des instances de composants offrant une de ces interfaces (ou une interface dérivée). Elles sont principalement mises en oeuvre par une instance de composant pour fournir de l'information, notification, soit au travers d'événements, soit par invocation directe. Leur utilisation peut être dynamique, grâce à l'utilisation de l'introspection sur les points de connexion, permettant la connexion et la déconnexion dynamique des composants consommateurs.

2.3.3.2 Mise en œuvre du modèle

L'implantation d'un composant COM se résume à la fourniture d'une bibliothèque offrant une ou plusieurs interfaces et implantant leur comportement. Ces bibliothèques se traduisent en général sous forme de DLLs (*Dynamically Linked Libraries*). La notion de conteneur pour héberger des instances de composants n'existe pas vraiment en COM, mais se retrouve en COM+. C'est en fait l'environnement, essentiellement MS-Windows, qui tient le rôle de conteneur. La création d'instances de composants se fait au travers du patron de conception [29] *Fabrique*. Une fabrique est offerte par un serveur de composant. Lors de l'insertion d'un serveur de composant dans un système, ce serveur est enregistré auprès d'un registre système défini par COM, qui maintient une liste des serveurs de composants disponibles.

La réutilisation en COM ne se traduit pas par l'héritage d'implantation (qui est possible en COM+, l'évolution de COM). Ceci s'explique par le fondement même de COM qui est de ne faire aucune spéculation sur l'implantation des composants (qui ne sont pas nécessairement des objets). Le seul héritage possible en COM est l'héritage d'interface. Une interface A hérite d'une interface B si l'interface A contient toutes les opérations définies dans B et en propose de nouvelles. En cela, le polymorphisme au sens de COM est le fait de proposer un ensemble d'interfaces.

Les deux modes de réutilisation d'implantations en COM sont la *composition* (appelée *containement*) et l'*agrégation* (voir la figure 2.13). Dans le cas de la composition, un composant dispose d'une référence exclusive sur un autre composant. Il s'agit de la réification de la notion de contenance. Le composant contenu (composant interne) est invisible des usagers qui ne sont conscients que du composant contenant (composant externe). Cette première possibilité est la méthode la plus simple et la plus courante (semble-t-il) pour faire de la réutilisation en COM.

Dans le cas de l'agrégation, il n'y a plus de notion de contenance entre les deux composants. Ils doivent en contrepartie collaborer. Un composant peut « choisir » s'il accepte de faire partie d'un agrégat ou pas. Pour ce qui est de la vision cliente de l'agrégat, toutes les interfaces de composants sont visibles au même titre. En effet, un agrégat se comporte comme un composant multi-interfaces. Les clients utilisent donc directement les composants constituant l'agrégat. L'utilisation de l'agrégation est un bon choix pour produire des *wrappers* de manière assez simple et automatique.

DCOM est une extension de COM qui prend en compte l'aspect distribué, principalement multi-processus, d'une application. Dans ce sens, DCOM définit la génération de *stubs* (les

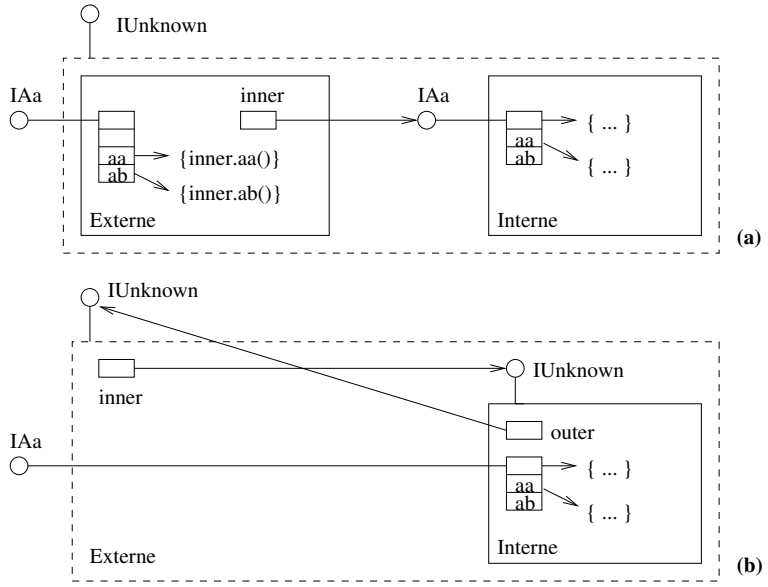


FIG. 2.13 – *Composition (a) et agrégation (b) avec COM*

squelettes du monde Microsoft) pour chaque interface d'un composant prenant en charge la partie communication avec un *proxy* (les souches du monde Microsoft). Ce *proxy* est utilisé par tout client du composant, qui pense utiliser une instance locale de composant. Pour la partie cliente, l'utilisation d'une instance de composant distante est transparente. La génération des *stubs* et *proxies* repose sur l'utilisation du langage de définition d'interfaces COM IDL.

COM+ est quant à lui une extension incluant un modèle d'objets légers (*lightweight object model*) à COM dans le but de prendre en compte des spécificités du langage Java. COM+ est un modèle uniquement utilisable dans un contexte intra-processus. Une utilisation inter-processus implique l'utilisation de DCOM. L'utilisation de COM+ étant fortement liée au langage Java, l'interface `IDispatch` disparaît de ce modèle (pour réapparaître sous la forme de l'API `reflect` de Java). Enfin, COM+ propose la notion de conteneurs offrant accès au moteur transactionel MTS (*Microsoft Transaction Server*) et prenant en charge les aspects sécurité et multi-threading.

2.3.3.3 Evaluation

Un apport important de COM est la possibilité de définir des **interfaces multiples sur un même composant**. Il est aussi possible de définir les **interfaces requises par un composant**. Il est donc dommage que la définition des interfaces d'un composant COM soit optionnelle. La définition des interfaces apporte à la fois une structuration et un support pour la collaboration entre les différents acteurs.

Les outils systèmes de COM pour la production d'applications se résument essentiellement à un environnement graphique de développement comme MS Visual Studio. Cet environnement fournit un ensemble de moyens pour produire une application : production de bibliothèque, d'exécutables, assistants pour la génération automatique de code, *etc.* En cela,

il n'est pas aisé de découper le processus de production en rôles distincts.

La diffusion d'implantations de composants se fait principalement au travers de bibliothèques (*DLL MS Windows*). Ces bibliothèques sont à déployer manuellement sur les machines cibles, selon une solution *ad hoc*. L'assemblage de composants est essentiellement réalisé à partir d'un environnement tel MS Visual Studio, la documentation des bibliothèques n'étant pas auto-contenue.

COM est une réponse de Microsoft à l'évolution du marché des applications. Microsoft fut un des premiers avec les modules Visual Basic à proposer une solution industrielle orientée composants. COM est donc une solution Microsoft pour l'environnement MS Windows (bien que des environnements COM semblent être disponibles sur d'autres plates-formes). Enfin, son utilisation n'est pas nécessairement simple : peu de spécifications des API, philosophie MS Windows imposée, principalement C++ comme langage de développement, *etc.* En conséquence, COM prend en compte uniquement l'étape d'implantation des composants. Les outils fournis ne permettent pas réellement de produire l'architecture d'une application, si ce n'est au travers de moyens de type programmation.

2.4 Modèles de référence

Les modèles de référence sont dans un sens à mi chemin entre les modèles académiques et les modèles industriels. Résultant d'un consensus entre industriels et académiques, on retrouve à la fois des innovations provenant du monde académique et la définition d'un cadre complet pour supporter la production industrielle d'applications.

2.4.1 Open Distributed Processing

La définition du modèle ODP (Open Distributed Processing) [38] a eu lieu lorsque les systèmes informatiques ont été amenés à gérer des environnements répartis utilisant des technologies hétérogènes, dont la configuration est vouée à évoluer progressivement, et dont tous les composants doivent pouvoir coopérer harmonieusement. En effet, il est nécessaire de s'affranchir des soucis d'hétérogénéité pour permettre l'interaction entre composants.

Cette situation appelle à définir des concepts et des règles partagés par tous les acteurs du développement des systèmes. Cette approche permet la conception de systèmes ouverts, qui sont conçus conformément à des standards reconnus par tous. Ces standards définissent les protocoles et interfaces que doivent posséder les systèmes, ils doivent être suffisamment abstraits pour permettre des implantations variées, mais assez directifs pour que des éléments réalisés séparément puissent coopérer.

Le modèle de référence ODP est le fruit de travaux menés durant plusieurs années à l'ISO (International Standardization Organization) et à l'ITU (International Telecommunication Union).

2.4.1.1 Modèle

La spécification d'un système réparti complexe gère de nombreuses informations. Une description rassemblant tous ces aspects est généralement irréaliste. Comme la plupart des

méthodes de spécifications, ODP prévoit donc un certain nombre de modèles (les points de vue), liés entre eux et qui sont des abstractions fournissant une spécification du système entier.

Le modèle ODP normalise différents points de vue, prescrit des modèles permettant de spécifier les principes et les règles de conformité adoptés dans chaque point de vue, et identifie les fonctions nécessaires à la réalisation d'applications réparties ouvertes. Cinq points de vue sont définis.

Le point de vue de l'information exprime la sémantique de l'application à l'aide de structures de données, de fonctions les manipulant et de propriétés d'intégrité sur ces données.

Le point de vue de l'entreprise exprime les objectifs, les droits et les obligations des entités qui composent une application.

Le point de vue des traitements exprime une vision fonctionnelle de l'application à des fins de répartition en définissant des entités fonctionnelles, en précisant leurs interactions et en déterminant les propriétés des fonctions correspondantes.

Le point de vue de l'ingénierie il décrit les fonctions et services de base que doit fournir l'infrastructure d'exécution pour que l'on puisse construire les mécanismes assurant l'inter-opérabilité en environnement hétérogène.

Le point de vue de la technologie expose les contraintes technologiques imposées par les mécanismes d'ingénierie à une infrastructure matérielle ou logicielle spécifique.

La définition du modèle de composants est présentée dans le point de vue traitement ; ainsi, nous ne discutons dans cette section que ce point de vue. Selon ce point de vue, une application est un ensemble d'objets, appelés *objets de traitement*, qui encapsulent un état interne et des données. Ils sont comparables à des composants. Un objet peut modifier son état interne en réalisant des traitements sur ses données. Une opération est un point d'accès à un traitement exécutable par un objet à la demande explicite de son environnement (c'est une méthode dans la terminologie objet).

Les opérations d'un objet sont regroupées dans des interfaces qui constituent l'unité de désignation et d'accès à un objet. Toute interaction avec un objet a lieu à travers une interface. On distingue deux types d'invocations, en fonction du modèle d'opération appelé. L'appel d'une opération sans terminaison est appelé une *annonce*, celui d'une opération avec terminaison est appelé *interrogation*. L'objet initiateur de l'invocation est appelé *client*, son destinataire est le *serveur*.

Seules les interfaces de service (décrivant les opérations offertes par un objet) sont explicitement décrites; un objet peut en posséder plusieurs, permettant ainsi de ne rendre accessibles que les sous-ensembles d'opérations qu'il peut traiter. Un objet A ne peut invoquer une opération sur un objet B que s'il dispose d'une interface de B décrivant cette opération. Les interfaces sont typées. Lorsqu'un client demande une interface d'un certain type, il peut obtenir la référence d'une interface non pas identique, mais conforme à celle qu'il a demandée. La conformité assure la transparence à la substitution : le client ne se rend pas compte que l'interface qu'il manipule n'est pas identique à celle qu'il a demandée. La conformité permet de vérifier la faisabilité des liaisons entre deux interfaces.

Le modèle de traitement ODP associe aux opérations des propriétés dites « transactionnelles », qui permettent d'exprimer les propriétés requises par les différents niveaux de cohérence. Ces propriétés concernent la visibilité (accessibilité des objets), la cohérence (invariants à satisfaire), la recouvrabilité (reprise sur erreurs), la permanence (irrévocabilité

d'une transaction réussie) et la dépendance (résistance aux erreurs d'autres transactions) des opérations.

Une *liaison* (voir figure 2.14) est l'abstraction des mécanismes mis en oeuvre pour assurer l'interaction entre deux objets lors d'une invocation d'opération. Elle doit assurer deux transparences essentielles, la transparence à la localisation, qui assure qu'un objet n'a pas à savoir où se situe l'objet qu'il invoque, et la transparence d'accès, qui garantit que l'invocation d'une opération se fera de façon identique, que l'objet invoqué soit distant ou local. Elle peut également assurer d'autres transparences : transparences de groupe, à la duplication, à la migration, aux pannes, transactionnelles, *etc.*

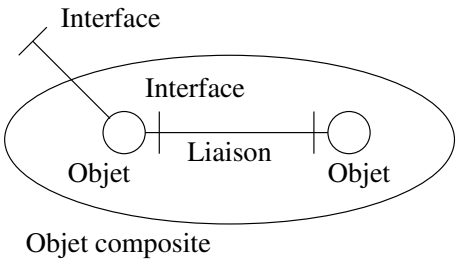


FIG. 2.14 – *Modèle de composants RM-ODP*

2.4.1.2 Mise en œuvre du modèle

Le point de vue ingénierie est complémentaire du point de vue traitement. Il a pour but de définir les constructions nécessaires à la mise en oeuvre d'un système ODP. Il fournit la spécification d'une machine virtuelle répartie afin de supporter le modèle défini lors du point de vue traitement. Ce point de vue prend en charge la réalisation des objets de liaison.

Mise en œuvre des composants du modèle de traitement La mise en oeuvre puis l'exécution des objets du modèle de traitement se traduisent par l'utilisation de ressources et la génération d'activités dans le système. L'unité d'encapsulation de l'exécution et des ressources est la capsule. L'infrastructure support doit offrir un service de gestion de capsules. A l'intérieur d'une capsule se trouvent des objets d'ingénierie de base, qui peuvent être regroupés en grappes (*clusters*). Une grappe est l'unité d'activation / désactivation / migration d'objets d'ingénierie de base.

Chaque noeud (unité physique localisée dans l'espace possédant des fonctions de calcul, mémorisation et communication) possède un noyau pour coordonner les fonctions du noeud afin de les rendre utilisables par les objets d'ingénierie (voir figure 2.15).

Mise en oeuvre des liaisons du modèle de traitement La mise en oeuvre d'une liaison est appelée un *canal* ; c'est un ensemble d'objets assurant la transparence à la distribution (accès et localisation) et gèrent le protocole de communication. La gestion de transparences additionnelles est obtenue en adaptant les gestions de configurations (choix de noeuds, de protocoles de communication, etc) et la gestion des ressources (dimensionnement des tampons, ordonnancement des tâches, etc).

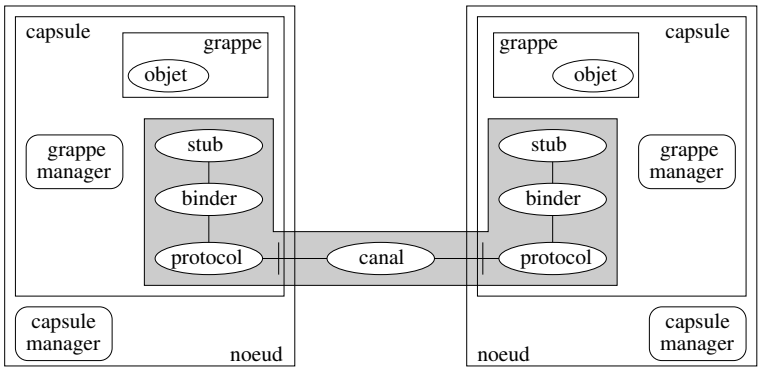


FIG. 2.15 – *Eléments du modèle d'ingénierie d'ODP*

2.4.1.3 Evaluation

Le modèle ODP propose une vision du cycle de développement des applications et, pour cela, fournit un modèle de haut niveau permettant l'analyse des besoins, la conception, l'implantation et l'assemblage des composants. Cette vision est structurée en mettant en œuvre la séparation des préoccupations au travers d'un découpage en points de vue du processus logiciel.

Le modèle de composant proposé identifie pour chaque composant ses interfaces ainsi que les **modes de communication** nécessaires pour chacun de ses services. Il supporte les **interfaces multiples, offertes et requises**. Un certain nombre de fonctions de transparence sont décrites, permettant d'**externaliser les aspects non fonctionnels** du cœur des composants. La dernière contribution du modèle ODP est la définition de **composants composites**, support à la structuration des applications.

Le modèle ODP est un modèle de référence qui doit être instancié. Il est décomposé en différentes parties qui proposent une méthodologie à suivre pour concevoir, déployer et utiliser des applications. En cela, ODP répond globalement aux étapes du processus de production des applications.

2.4.2 CORBA Component Model

L'utilisation d'objets répartis avec la technologie CORBA [32, 71] de l'Object Management Group (OMG) n'a pas permis d'atteindre la simplicité escomptée pour concevoir des applications distribuées à base d'entités logicielles hétérogènes et multi-fournisseurs. Pour faciliter et augmenter la qualité du processus de production de telles applications, l'OMG a défini le CORBA Component Model (CCM) [50, 48, 69], un modèle de composants logiciels serveurs répartis et hétérogènes, reposant sur une technologie de conteneurs similaire à celle des EJBs.

2.4.2.1 Modèle

Le modèle de composants CORBA repose sur deux points essentiels. Premièrement, un type de composant décrit l'extérieur d'une boîte noire dont on ne sait rien de l'implantation.

Deuxièmement, seule l'implantation fonctionnelle devrait être à programmer et tout le reste doit être décrit. Le premier point met en avant la volonté de rendre explicite toute interaction avec ou à partir du type de composant. Dans ce sens, un type de composant ne se limite pas à définir les services qu'il fournit, mais aussi les services qu'il utilise. Le second point tend à faciliter la production et à accroître la réutilisabilité des implantations de composants. La description sert de base à la prise en charge des aspects non fonctionnels.

Dans le cadre du modèle abstrait de composants tel qu'illustré dans la figure 2.16, un composant est potentiellement multi-interfaces. La partie gauche de la figure présente les différentes interfaces fournies par un type de composant. Ces interfaces peuvent être de deux types : des facettes, qui offrent un mode de coopération synchrone, et des puits d'événements, qui offrent un mode de coopération asynchrone. Chaque facette regroupe une partie des opérations disponibles sur le type de composant, et sont sémantiquement équivalentes à des vues sur le type de composant. A partir de la référence de base d'une instance de composant, il est possible de *naviguer* entre les différentes facettes au cours de l'exécution.

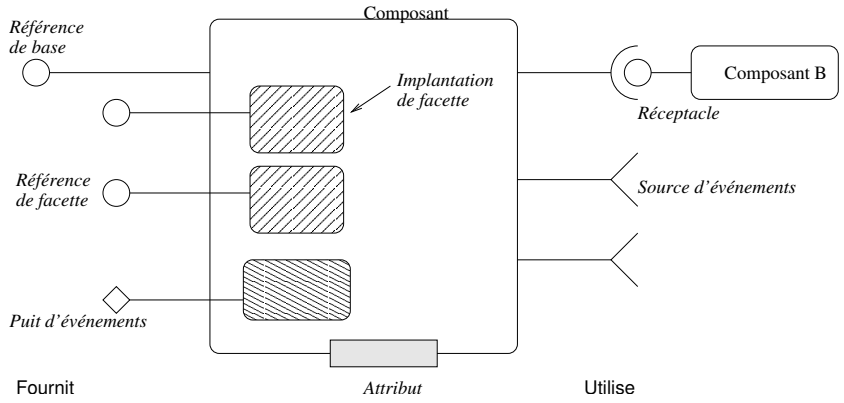


FIG. 2.16 – *Modèle abstrait du CORBA Component Model*

La partie droite de la figure 2.16 présente les interfaces utilisées par un type de composant. De même que pour les interfaces fournies, deux types de coopération sont disponibles : l'utilisation synchrone de types de composants tiers au travers des réceptacles, et l'utilisation asynchrone au travers des sources d'événements. L'utilisation d'événements se fait selon le modèle « publish / subscribe » de CORBA. Les communications asynchrones peuvent être de deux types : $1 - vers - 1$, coopération privée entre deux instances de composants, ou $1 - vers - n$, coopération mettant en jeu plus de deux instances de composants.

Les implantations de facettes et de puits d'événements de la figure 2.16 représentent l'implantation fonctionnelle d'un composant. Cette partie de l'implantation est la seule à être programmée. Le reste de l'implantation comme la prise en charge des points de connexion (aussi appelés *ports*), et de l'aspect persistance d'un composant sont décrits à l'aide du langage CIDL (Component Implementation Definition Language), et générés automatiquement.

Pour chaque type de composant est défini un (ou plusieurs) type de *maison de composant*. Une maison de composant est un gestionnaire d'instances construit autour des deux patrons de conception [29] *Fabrique* et *Recherche*. Ce gestionnaire prend en charge le cycle de vie des instances de composants.

2.4.2.2 Mise en œuvre du modèle

Pour permettre la définition des types de composants, le langage OMG IDL a été étendu afin de prendre en compte les nouveaux concepts comme les ports. Cette extension permet de définir, comme illustré sur la figure 2.17, les différentes facettes (mot-clé `provides`), les puits d'événements (mot-clé `consumes`), les réceptacles (mot-clé `uses`) ainsi que les sources d'événements (mot-clé `emits` pour une coopération $1 - vers - 1$ et `publishes` pour une coopération $1 - vers - n$).

Le type de maison `DistributeurHome` est déclaré comme prenant en charge le type de composant `Distributeur`. L'opération de base `create()` sera automatiquement générée pour permettre la création d'instances de composants de ce type. Cette description en IDL étendu n'est utilisée que pour exprimer le résultat de la conception. Pour être utilisée, cette déclaration est projetée en IDL telle que défini dans la spécification CORBA 2. Les habitudes des développeurs ne sont ainsi pas modifiées.

```
component Distributeur {
    provides FacetteClient client;
    provides FacetteFournisseur fournisseur;
    consumes TemperatureEvt temperature;
    uses PriseCourant courant;
    emits VideEvt vide;
}

home DistributeurHome manages Distributeur { };
```

FIG. 2.17 – *Définition d'un type de composant CORBA et d'un type de maison*

Afin de permettre une bonne intégration des parties fonctionnelles et non fonctionnelles de l'implantation d'un composant, le CCM fournit un canevas, le *Component Implementation Framework* (CIF), qui spécifie comment les deux parties de l'implantation doivent coopérer. Ce canevas fixe d'autre part la façon dont est générée la partie non fonctionnelle. Le CIF s'appuie sur un langage déclaratif, le *Component Implementation Definition Language* (CIDL), qui permet de décrire un composant et diriger la génération. Actuellement, seulement le type (service, session, etc.), les besoins en persistance et la structure d'implantation sont exprimés.

Comme dans le cadre des EJBs, les composants CORBA s'exécutent dans un conteneur qui leur offre d'une part un environnement d'exécution (espace mémoire, flot d'exécution), et d'autre part les services systèmes nécessaires. La complexité de ces derniers est masquée au développeur d'implantations de composants, qui ne se préoccupe plus de maîtriser ces aspects. Les conteneurs se déclinent selon différents types de base, mais sont génériques par rapport à une même famille de composants (service, session, processus ou entité).

Les échanges entre instances de composants et conteneurs sont définis par des APIs standardisées. Le conteneur offre les interfaces des aspects système, comme la persistance et les aspects transactionnels, aux instances de composant ; les instances de composants disposent d'une interface de rappel permettant au conteneur d'agir sur elles (principalement pour gérer la persistance de ces instances de composant).

Dans le but d'automatiser au maximum la phase de déploiement des applications à base

de composants CORBA, la spécification étend le langage Open Software Description (OSD) pour fournir les informations nécessaires au déploiement. Ce langage spécifié par le W3C et étendu par l'OMG est défini comme un vocabulaire XML (eXtensible Markup Language). Il permet de décrire d'une part les besoins des implantations de composants (persistance, sécurité, transactionnel) et d'autre part les assemblages de composants. L'architecture d'une application est ainsi décrite de manière abstraite et pourra être projetée sur un ensemble de ressources physiques lors du déploiement. Nous détaillons ce dernier point dans la section 3.4.3).

2.4.2.3 Evaluation

Tout comme ODP, le CCM est un modèle dont les spécifications sont rendues disponibles, mais pour lequel aucune implantation de référence n'existe. Un fournisseur doit se conformer à la spécification pour fournir un environnement CCM. Actuellement, aucun environnement n'est disponible dans un contexte industriel. Des plates-formes sont en cours de réalisation [49, 105, 106], dont notre plate-forme OpenCCM [53] visant à devenir la plate-forme de référence disponible en logiciel libre. Cette étude s'est appuyée à la fois sur la spécification et sur les travaux relatifs à l'implantation de notre plate-forme, qui a validé un certain nombre de concepts du CCM.

Un des apports essentiels du CCM est l'ensemble des moyens fournis pour décrire les types, les implantations et les assemblages de composants. Le CCM fait une bonne synthèse du **concept de port**, de la possibilité de définir des **interfaces multiples** et des **interfaces requises**. Cette proposition fait un grand pas vers la non-programmation des applications : décrire au mieux pour programmer au minimum. Cette puissance d'expression prend tout son sens sur les **aspects non fonctionnels des composants** qui peuvent, pour une même implantation, varier en fonction du contexte d'exécution. Le second atout majeur des composants CORBA est le fait d'**expliciter les dépendances des composants**, en termes de composants et de ressources système, pour permettre une composition dynamique des instances. Ensuite, le modèle de déploiement est une première réponse encourageante vers une automatisation du processus de déploiement des applications distribuées. Enfin, le CCM est la seule proposition qui veut permettre la construction d'applications à l'aide de **composants hétérogènes** et d'outils multifournisseurs.

Cependant, le CCM n'est pas une technologie utilisable à l'heure de l'écriture de ce document. La spécification souffre encore de quelques défauts de jeunesse, principalement une sous-spécification de certains aspects (comme la définition de la structure d'un composant en CIDL et l'architecture des environnements d'exécution des composants). La version actuelle, représentant plus de 500 pages, est complexe à maîtriser dans son ensemble, complexité qui ne devrait pas retomber sur le développeur, mais qui explique en partie l'absence d'implantations. Une description plus complète de celle-ci est disponible dans [50, 48]. Les travaux menés actuellement à l'OMG au sein de la *Revision Task Force* vont dans le sens où le CCM sera utilisable en prenant en compte l'ensemble des étapes du processus de production. D'autre part, même si l'interopérabilité en termes d'acteurs sera limitée dans un premier temps, la version 2.0 du CCM devrait avoir comme objectif principal de fournir des réponses à cette attente.

2.5 Conclusion

Les différents modèles de composants présentés dans ce chapitre mettent l'accent sur un certain nombre de concepts et en mettent d'autres de côté. De notre point de vue, un modèle « idéal » devrait supporter un certain nombre de concepts.

Interfaces multiples Un composant peut être utilisé au travers de différentes interfaces distinctes.

Interfaces requises Un composant précise les interfaces fournies par d'autres composants qu'il utilise pour réaliser ses traitements.

La notion de port Un composant fournit des mécanismes pour être connecté, éventuellement dynamiquement, à d'autres composants. Un port correspond à une interface fournie ou requise.

Synchrone Un composant supporte le mode de communication synchrone, tel l'invocation d'opérations, avec des composants tiers.

Asynchrone Un composant supporte le mode de communication asynchrone, tel l'émission et la réception d'événements, avec des composants tiers.

Connecteur Les connexions entre les composants sont réifiées à l'exécution et manipulables.

Composite Les composants peuvent être hiérarchiques, c'est-à-dire composés d'autres composants. Les composites sont manipulables comme des composants.

Conteneur Un conteneur fournit un environnement d'exécution aux composants, un accès simplifié aux services de base et peut prendre en charge un certain nombre d'aspects techniques comme la communication.

Non fonctionnel Les propriétés non fonctionnelles des composants ne sont pas mélangées au code métier, et sont prises en charge par l'environnement (par exemple, le conteneur).

Package Un package de composant regroupe interfaces, implantations et description d'un composant. Il permet sa diffusion et contribue à l'automatisation de son déploiement.

Concepts / modèles	Darwin	JavaPods	JavaBeans	EJB	(D)COM	ODP	CCM
Itf multiples					x	x	x
Itf requises	x		-		x	x	x
Notion de port	x					x	x
Synchrone	x	x	x	x	x	x	x
Asynchrone	x	x	x	x	-	x	x
Connecteur		x				x	
Composites	x	-				x	
Conteneurs		x	-	x	-	x	x
Non fonctionnel		x		x	-	x	x
Packages		x	x	x			x

FIG. 2.18 – *Synthèse des concepts présents dans les modèles de composants*

Le tableau de la figure 2.18 synthétise le support de ces différents concepts par les modèles de composants que nous avons présentés dans ce chapitre. La présence d'un « x » signifie que

le concept est bien supporté par le modèle de composant. La présence d'un « - » signifie que le concept est faiblement supporté. Ce tableau montre essentiellement qu'il n'existe pas de modèle de composant « idéal ». L'utilisation d'un modèle de composant particulier résulte donc d'un choix qui doit être guidé par les besoins applicatifs. Les applications de commerce électronique sont en général des applications à trois niveaux qui trouvent une bonne réponse dans les EJBs : le niveau du milieu offre des services et utilise directement des bases de données. Les applications de télécommunication sont quant à elles très réparties et requierent des modèles de composants comme le CCM : un composant offre des services à n composants et utilise les services de m autres composants.

Bien que les composants ne soient pas la réponse ultime aux problèmes rencontrés par l'ingénierie du logiciel, ils représentent une bonne réponse au moins en partie, et sont de mieux en mieux acceptés et utilisés dans l'industrie du logiciel. Ils sont essentiellement appréciés dans le contexte du logiciel sur l'étagère, un composant logiciel est produit de sorte à être réutilisé dans différentes applications. Au delà de ce constat, les composants apportent à la fois une structuration des applications et permet, de par leur nature, d'améliorer la collaboration des acteurs d'un processus logiciel. Les composants représentent une bonne réponse à la définition et à la fourniture des briques de base des applications.

Toutefois, **les composants représentent uniquement une vision microscopique des applications**, leurs briques de base. Les modèles de composants n'offrent que des **solutions très limitées pour exprimer une vision macroscopique des applications**, leur architecture. Les composants ne répondent donc que partiellement aux besoins d'un processus logiciel. Dans le but de produire des applications, les modèles de composants doivent être associé à un moyen permettant la définition d'architectures logicielles. Le chapitre suivant va discuter des langages de description d'architecture. Nous parlerons des éléments importants fournis par ces langages, et soulignerons en quoi ils complètent les modèles de composants comme support de la mise en œuvre d'un processus d'ingénierie du logiciel.

Sur un autre plan, il est important de prendre un peu de recul sur les composants. A regarder leur évolution depuis 1996 [1], force est de constater que les composants sont un domaine en pleine évolution, qui plus est rapide. En effet, le paysage industriel était à l'époque réduit aux JavaBeans et à l'utilisation de Visual Basic pour intégrer des DLL Windows ou de langages de scripts pour intégrer des librairies partagées sous Unix. Après l'apparition des EJBs en 1997, le mouvement s'est accéléré pour arriver aujourd'hui à des modèles beaucoup plus complets et intéressants, même si malheureusement plus complexes globalement, tels que les EJBs 2.0 ou le CCM. Il est donc intéressant d'utiliser les modèles de composants, mais il est surtout **important de ne pas s'enfermer dans un modèle technologique**, tout d'abord car le meilleur est à venir et ensuite parce que les besoins vont irrémédiablement évoluer. Il nous semble donc souhaitable de structurer les applications à l'aide de composants tout en exprimant leur architecture indépendamment de tout modèle technologique, pour à la fois bénéficier des modèles actuels et pouvoir dans l'avenir évoluer vers des modèles plus complets. D'un point de vue technologique, le CCM nous semble actuellement le plus complet et c'est pourquoi nous l'avons choisi pour nos premières réalisations sur ce modèle [50].

1. La première utilisation du terme composant logiciel semble remonter à la conférence de l'OTAN sur l'ingénierie du logiciel en 1968.

Chapitre 3

Langages de description d'architectures

Ce chapitre présente une vision macroscopique du processus logiciel. Les langages de description d'architectures (ADLs, *Architecture Description Language*) représentent actuellement le meilleur support à la définition d'architectures logicielles. Une architecture définit l'ensemble des composants constituant une application ainsi que leur interconnexion. Au delà de ces deux points, les ADLs peuvent avoir des objectifs variés. Nous allons discuter dans ce chapitre des réponses apportées par ces langages aux préoccupations des processus logiciels.

La section 3.1 rappelle tout d'abord une définition d'architecture logicielle, ainsi que les concepts sous-jacents aux langages de description d'architectures. La suite de ce chapitre présente certains de ces langages. Cet ensemble a été choisi comme illustration du panel de préoccupations que l'on peut trouver dans ces langages, qui peuvent être regroupés en trois grandes familles.

- La section 3.2 présente Rapide et Wright qui sont des langages formels. Leur motivation principale est de capturer le comportement des applications à des fins de vérification automatisable.
- La section 3.3 présente ACME et xArch. Ces langages que nous qualifions de « partage » visent à l'échange et l'intégration d'éléments architecturaux définis à l'aide de langages différents.
- La section 3.4 présente les langages de configuration C2, Olan, et les *Component Assembly Descriptors* du modèle de composants CORBA. La motivation de ces propositions est d'exploiter les descriptions d'architectures dans le but de générer en partie les composants logiciels ou de supporter l'automatisation du processus de déploiement des applications.

Enfin, la section 3.5 présente une synthèse des langages de description d'architectures. Elle dresse également une vision synthétique des différentes préoccupations prises en compte dans les langages présentés dans ce chapitre.

3.1 Concepts sous-jacents aux architectures

3.1.1 Définition

Dans [87], Mary Shaw *et al.* définissent une architecture logicielle de la manière suivante :

> *The architecture of a software system defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence*

> *between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, and component compatibility. Architectural models clarify structural and semantic differences among components and interactions. Architectural definitions can be composed to define larger systems. [...]*

De cette définition, nous pouvons déjà mettre en avant le fait que l'architecture est une vision d'ensemble, qui plus est abstraite, des applications. Ceci représente déjà un élément pour considérer les modèles de composants et les ADLs comme complémentaires. De plus, nous pouvons nous élever par rapport à notre processus d'ingénierie du logiciel et le considérer à un niveau macroscopique : les architectures d'applications. Il faut cependant garder à l'esprit que ce n'est pas parce que deux propositions semblent complémentaires que leur utilisation le sera. Il est donc important de considérer comment le lien entre ADLs et modèles de composants est mis en œuvre.

Nous reprenons ici les définitions des concepts sous-jacents aux ADLs (composant, connecteur et configuration) tels que présentés par Medvidovic et Taylor dans [56] ainsi que W. Ellis *et al.* dans [27]. La caractérisation d'un composant logiciel présentée ici est moins complète que la caractérisation discutée dans le chapitre 2, mais repose sur les ADLs étudiés et leur vision de ce concept.

3.1.2 Composant

Le *composant* est présenté de manière grossière dans le cadre des ADLs comme une unité de traitement ou de stockage de données. Dans le cadre des différents ADLs discutés, la granularité des composants varie de la définition d'une simple fonction à la définition d'une application dans son intégralité. Medvidovic et Taylor caractérisent les composants selon cinq critères : interface, type, sémantique, contraintes et évolutions. Les propriétés non fonctionnelles des composants ne sont pas prises en compte à ce niveau. Par rapport à leur discours, ces dernières sortent du champ de leurs motivations.

Comme déjà discuté dans le chapitre 2, les interactions entre un composant et le monde extérieur sont définies au travers d'une ou plusieurs interfaces. Cette interface décrit les services fournis et les services requis par un composant. Ces services peuvent être définis comme des signatures de méthodes, des messages ou encore des variables. Les différents ADLs présentés ici manipulent le concept d'interface, mais selon leur propre point de vue. Par exemple, dans le cas d'ACME [31] ou de Wright [4] chaque interface fournie / requise est un port, alors que pour C2 [95, 57] chaque composant dispose d'une unique interface. Enfin, dans le cas de Rapide [96], une interface est appelée *constituant*.

Capturer la structure d'une application est la motivation première des langages de description d'architectures. La réutilisation du logiciel en est une seconde [30]. Pour cela le concept de type de composant est utilisé et distingué des instances, afin de pouvoir définir et représenter une architecture sous forme abstraite. De plus, l'utilisation de types de composants permet de définir un ensemble d'instances ayant des propriétés en commun, ce qui introduit la substituabilité des instances. Les différents ADLs dont nous discutons dans ce chapitre font la distinction entre type et instance de composant.

La définition d'interfaces répond de manière très élémentaire à l'expression de la sémantique des composants. Pour permettre la vérification des contraintes ou encore le test / la simu-

lation des architectures, il est nécessaire de disposer d'un modèle définissant la sémantique des composants. Par exemple, Rapide repose sur l'ordonnancement partiel d'événements, et définit la sémantique comportementale [5].

3.1.3 Connecteur

Les *connecteurs* sont des éléments architecturaux modélisant les interactions entre composants, ainsi que les règles associées. La complexité d'un connecteur peut varier d'un simple appel de procédure à distance (RPC, *Remote Procedure Call*) [9] à la mise en œuvre d'un protocole de communication sécurisé et transactionnel. Dans le cas des ADLs présentés ici, les connecteurs peuvent être modélisés explicitement, comme dans Wright et ACME où ils sont désignables et manipulables, ou alors implicites comme dans le cas de Rapide ou de ArchJava [2, 3]. Pour que des connecteurs soient réutilisables et extensibles, il est nécessaire qu'ils soient considérés comme des entités de première classe, *i.e.* au même titre que les composants.

Dans le cas d'un connecteur modélisé explicitement, il y a une forte similitude avec la définition de composants. L'interface du connecteur définit le rôle des différents participants à l'interaction. Ces interfaces ne décrivent pas des services fonctionnels, mais les mécanismes de connexion entre composants. Ensuite, la description de l'implantation du connecteur définit le protocole associé à l'interaction qu'il est nécessaire de mettre en œuvre. Cette implantation n'est pas nécessairement réifiée en une seule entité ; c'est le cas dans les RPCs où le connecteur correspond à la fois à la souche cliente et au squelette serveur.

Toujours dans le but de supporter l'analyse des interactions entre composants et la vérification des contraintes imposées sur les composants et les connecteurs, la sémantique associée à ces derniers doit être spécifiée. En règle générale, les ADLs utilisent le même modèle sémantique pour les composants et les connecteurs. Ici encore, l'utilisation uniquement d'interfaces est une solution n'offrant que peu de sémantique. Rapide, au travers des *posets* (*Partially Ordered Set of Events)*, et Wright, au travers du langage CSP, permettent d'exprimer la sémantique des connecteurs.

3.1.4 Configuration

Une *configuration architecturale* définit la structure de l'architecture d'une application au travers d'un graphe de composants et de connecteurs. Cette information permet de valider le bon assemblage des composants et connecteurs en terme d'interfaces fournies et requises, ainsi que de vérifier le comportement global d'une architecture. Associée aux modèles des composants et des connecteurs, la définition de configuration permet d'évaluer les aspects liés à la répartition d'une architecture comme les risques d'interblocage ou de performance.

La définition de l'architecture doit permettre de comprendre au mieux une application ou un système, d'en fournir une abstraction. Il est donc important pour un ADL de fournir une vision simple et compréhensible en soi. Pour cela, l'expression explicite de configurations offre la meilleure compréhension possible d'une architecture. Un second aspect intéressant, qui était absent des modèles de composants du chapitre 2, est la *hiérarchisation* d'une configuration. Ceci permet de voir une même configuration à différents niveaux d'abstraction, offrant tous les détails possibles ou bien un simple composant à inclure dans une configuration encore plus large. Pour cela, ACME offre les *templates*, C2 les *composants internes* d'architectures, Olan

les *composites*, alors que Wright permet les configurations hiérarchiques sans fournir de moyens explicites. Cette capacité tend à faciliter le passage à l'échelle et l'évolution des applications produites. Enfin, certains ADLs proposent des *styles d'architectures*, qui représentent des *patterns* de composition, pour faciliter la mise en place de configurations.

La définition d'architectures prend tout son sens dans le cadre des applications de grande taille. Dans ce contexte, il est, et sera, de plus en plus courant de faire intervenir un grand nombre d'acteurs et une variété d'environnements d'exécution, de modèles de composants et de langages de programmation. Il est donc important que les langages de descriptions d'architectures permettre de construire des applications dans un contexte fortement hétérogène. Une configuration doit donc pouvoir intégrer des composants et des connecteurs de granularité et de modèles variés. D'autre part, une application de grande taille peut difficilement être arrêtée pour des actions telles que la reconfiguration ou l'ajout de nouvelles fonctionnalités, et il est important de pouvoir exprimer un certain dynamisme des configurations. Il est donc souhaitable de trouver, comme dans C2, le support pour l'ajout, le retrait et la reconnexion de composants à l'exécution.

3.2 Langages formels de description d'architectures

Cette section présente deux langages formels de description d'architecture : Rapide et Wright. L'objectif principal de ces langages est de capturer le comportement des applications dans le but d'automatiser leur vérification.

3.2.1 Rapide

3.2.1.1 Modèle

Rapide [96] est le résultat d'un travail commun entre l'université de Stanford et la compagnie TRW inc. Sa préoccupation principale est la vérification des architectures logicielles au travers de la simulation. Pour cela, *Rapide* permet de définir une architecture comme un assemblage de modules et de composants communicants par échange de messages au travers d'événements.

Toute information transmise est considérée comme un événement, que ce soit une demande de service ou la valeur à donner à un attribut d'un composant. Les patrons d'événements (*event patterns*) permettre de définir les interactions entre composants et donc le comportement des applications, en caractérisant les événements mis en jeu. Un patron d'événement est défini à l'aide d'opérateurs exprimant leurs dépendances, comme la dépendance causale ($\rightarrow$), l'indépendance ($||$) et la simultanéité (*and*).

Rapide permet de définir un composant à l'aide d'une interface spécifiant les services fournis et requis, ainsi que le comportement du composant. Le comportement reflète le fonctionnement observable, c'est-à-dire l'ordonnancement des messages et appels aux services. Les patrons d'événements permettent de spécifier les contraintes d'un composant, c'est-à-dire les règles relatives à l'enchaînement des événements reçus et émis. Trois types de services sont disponibles sur les composants :

- *Provides* reflète les services synchrones fournis par le composant ;
- *Requires* reflète les services synchrones requis par le composant ;

– *Actions* reflète les échanges asynchrones entre composants, *in* pour les événements recevables et *out* pour les événements émissibles.

3.2.1.2 Mise en œuvre

Une définition d'architecture, appelée configuration dans d'autres ADLs, contient la définition des composants mis en jeu et leurs règles d'interconnexion. Toute instance est déclarée au même titre qu'une variable dans un langage de programmation. Une règle d'interconnexion est définie à la fois par un patron d'événement à vérifier et par un patron d'événement à déclencher en cas de succès. Tout comme un composant, une architecture peut contenir des contraintes, toujours des patrons d'événements (qui spécifient le comportement global de l'architecture) en précisant le comportement de certaines connexions.

La spécification d'une connexion se fait en mettant en relation deux patrons d'événements correspondant respectivement aux services requis et fournis par les composants mis en jeu. Trois types d'opérateurs de connexion sont disponibles.

– *To* connecte deux patrons simples, c'est-à-dire mettant en relation uniquement deux composants (un émetteur et un récepteur). Dans le cas où le patron de gauche est vérifié, le patron d'événement de droite déclenche l'événement associé sur le composant récepteur.

– `|>` connecte deux patrons quelconques, c'est-à-dire pouvant spécifier plusieurs récepteurs. Si la partie droite de l'expression est vérifiée alors tous les composants concernés par la partie droite reçoivent les événements du patron. Dans le cas de cet opérateur, l'ordre d'évaluation est quelconque, c'est-à-dire indépendant des évaluations antérieures ou postérieures. Cet opérateur est dit de *diffusion*.

– `=>` ajoute la notion d'ordre d'évaluation à l'opération de connexion précédente, *i.e.* un déclenchement d'une règle ne sera effectif que lorsque les déclenchements précédents de cette règle auront été évalués. Cet opérateur est appelé *pipeline*.

La figure 3.1 présente une définition du dîner des philosophes en *Rapide*. La règle d'interconnexion entre un philosophe et sa fourchette présentée ici précise qu'un philosophe peut demander l'acquisition d'une fourchette. L'utilisation de l'opérateur *pipeline* précise que sa requête ne pourra aboutir que lorsque les requêtes émises antérieurement sur cette fourchette par ses confrères auront été évaluées.

```
with Philosophe, Fourchette ;

?p : Philosophe ;
!f : Fourchette ;
?m : Message ;

?p.Send (?m) => !f.Receive (?m) ;
```

FIG. 3.1 – *Exemple de définition d'architecture avec* Rapide

3.2.1.3 Evaluation

Rapide représente une réponse intéressante à la spécification et à la **vérification du comportement d'une architecture logicielle**. En effet, en plus de spécifier l'architecture, *Rapide* permet de vérifier son comportement à l'aide de l'environnement de simulation associé. Cette fonctionnalité est propre aux ADLs que nous avons qualifié de formels et ne se retrouve pas dans les autres ADLs que nous présentons ici.

En contrepartie, les préoccupations telles que le développement, le déploiement et l'exécution d'applications ne sont pas prises en compte par *Rapide*. Il n'est pas possible de générer, même en partie, l'implantation d'une application définie avec *Rapide*. Seuls les concepteurs d'applications et architectes y trouveront donc un support à leur activité.

Cependant, même pour les architectes, le fait que les connecteurs ne soient pas explicites en *Rapide* ne facilite pas leur réutilisation. Un connecteur n'est spécifié que par les interfaces des composants mis en jeu. Enfin, il n'est pas possible de spécifier les propriétés non fonctionnelles des composants. Mais cette dernière remarque est liée à la préoccupation première de *Rapide* qui n'est pas de développer mais de **simuler une architecture fonctionnelle**.

3.2.2 Wright

3.2.2.1 Modèle

Wright [4] est un langage de description d'architecture développé à la Carnegie Mellon University de Pittsburg. Tout comme *Rapide*, *Wright* fait partie des ADLs formels, et sa préoccupation principale est la spécification d'architectures. En plus des trois concepts de base des ADLs présentés dans la section 3.1, *Wright* définit aussi la notion de style.

Dans le cadre de *Wright*, un composant est une entité de traitement abstraite indépendante. Tout comme dans la majorité des modèles, un composant dispose d'une interface définissant ses interactions possibles au travers de ports. Comme dans le cas du CCM (voir section 2.4.2), les ports sont à comprendre comme des facettes sur les composants. En parallèle de son interface, les traitements associés à un composant peuvent être spécifiés. Le comportement de chaque port est spécifié formellement et le comportement du composant est décrit en fonction du comportement de chaque port.

Comme pour *Rapide*, les connecteurs de *Wright* permettent de spécifier des interactions entre plusieurs composants. Toutefois, *Wright* permet de typer les connecteurs et donc de motiver leur réutilisation. Un connecteur est exprimé de manière explicite et abstraite et représente un patron d'interactions. Les connecteurs définissent les rôles auxquels peuvent participer le connecteur et la glue qui va permettre de lier rôles et participants. Un rôle définit le comportement de certains composants au sein de l'interaction.

Dans le cadre de *Wright*, un style permet de définir une famille d'applications au travers de propriétés communes. Le principe de base d'un style est de définir une ontologie, soit un vocabulaire commun à toutes les architectures de la famille d'applications. L'ontologie spécifie un ensemble de types de composants, de connecteurs et de propriétés courants dans ces architectures.

```
Configuration Diner
  Component Philosophe
    Port gauche
    Port droite
    Computation (boucle {acquerir gauche et droite,
                 manger, libérer gauche et droite
                 penser})

  Component Fourchette
    Port prendre (acquérir la fourchette)
    Computation  (la fourchette se comporte comme
                  un sémaphore)

  Connector Main
    Role proprio (le philosophe qui veut manger)
    Role saisir  (la fourchette attribuée)
    Glue

  Instances
    p1, p2 : Philosophe
    f1, f2 : Fourchette
    m1, m2, m3, m4 : Main

  Attachements
    p1.gauche as m1.proprio
    m1.saisir as f1.prendre
    p1.droite as m2.proprio
    m2.saisir as f2.prendre
    p2.gauche as m3.proprio
    m3.saisir as f2.prendre
    p2.droite as m4.proprio
    m4.saisir as f1.prendre
End Diner
```

FIG. 3.2 – *Exemple de définition d'architecture avec* Wright

3.2.2.2 Mise en œuvre

Tout comme dans *Rapide*, la définition d'une configuration regroupe les types (composants et connecteurs), les instances (à la fois de composants et de connecteurs) qui se retrouveront à l'exécution, et l'interconnexion des composants à l'aide des connecteurs. Dans le contexte de *Wright*, une configuration peut être hiérarchique. Dans ce cas, la définition d'un composant composite faisant partie d'une configuration globale contient la configuration associée à sa composition.

Les spécifications de comportement sont exprimées à l'aide du langage formel CSP au travers d'un processus (*CSP process*). Comme dans le cas des patrons d'événements de *Rapide*, un processus définit un ensemble d'événements observables et potentiellement déclenchés par celui-ci. Dans la syntaxe CSP, les noms d'événements déclenchables par le processus sont surlignés (ce n'est pas le cas des événements observables). Tout événement peut émettre des données (`evt!data`) ou en recevoir (`evt?data`). Tout comme les configurations, les processus CSP peuvent être hiérarchiques et les compositions de processus soit séquentielles soit exclusives (à rapprocher de l'ordonnancement dans le cas de *Rapide*).

La figure 3.2 présente une version simplifiée du dîner des philosophes en *Wright*. Cette configuration définit deux types de composants, `Philosophe` et `Fourchette`, et un type de connecteur, `Main` (les mains des philosophes). Pour ne pas présenter un exemple trop long, deux instances de philosophes et de fourchettes, ainsi que quatre connecteurs sont définis. La partie *Attachements* définit comment les philosophes et les fourchettes sont connectés au travers des mains des philosophes. Les informations contenues entre parenthèses représentent la spécification du comportement des composants, dans une forme proche du français. Il n'y a toutefois pas de formalisme pour exprimer cette sémantique.

3.2.2.3 Evaluation

L'apport principal de *Wright* est la fourniture d'un langage formel de spécification (CSP) pour exprimer, en plus de leurs interfaces, **le comportement des interactions entre les composants et les connecteurs**. Ce langage peut servir de support à l'échange, sous forme textuelle, entre acteurs du processus d'ingénierie du logiciel (le formalisme commun). La présence de deux modèles distincts pour définir les types de composants et de connecteurs permet une spécification indépendante de ces deux catégories d'entités.

Parallèlement, on peut reprocher à *Wright* de ne pas fournir un modèle commun pour les différents éléments manipulés. L'**expression de la sémantique liée au comportement des composants et des connecteurs** représente un intérêt majeur dans l'utilisation de *Wright*. Toutefois, il est regrettable que cette sémantique ne soit qu'exprimée et non exploitée de manière systématique. En effet, *Wright* ne fournit ni simulateur, contrairement à *Rapide*, qui permettrait de valider une architecture, ni générateurs de code, support de la conformité d'une implémentation avec son expression en *Wright*.

Nous trouvons donc encore dans cet ADL une réponse adéquate aux préoccupations des concepteurs et des architectes, mais une réponse partielle aux développeurs : le formalisme commun. Enfin, les autres acteurs (intégrateur, placeur, déployeur et administrateur) ne trouveront pas de support à leurs préoccupations dans le contexte de *Wright*.

3.3 Langages d'échange ou d'intégration d'architectures

Cette section présente deux langages d'échange ou d'intégration d'architectures : ACME et xArch. Ces langages ont pour but le partage ou l'intégration d'éléments architecturaux définis à l'aide d'ADLs hétérogènes.

3.3.1 ACME

3.3.1.1 Modèle

ACME [31] est aussi le résultat de travaux menés à l'université de Carnegie Mellon à Pittsburg, mais, contrairement à *Wright*, ces travaux sont communs avec l'*USC/Information Science Institute*. L'objectif principal d'*ACME* n'est pas de fournir un nouvel ADL, mais un langage pivot pour fédérer les ADLs existants. Le but est de pouvoir intégrer des définitions réalisées avec différents ADLs pour en favoriser l'échange. Indirectement, *ACME* peut servir de glue quant à l'exploitation conjointe des avantages de chacun des ADLs intégrés. *ACME* se focalise donc sur la syntaxe et non sur la sémantique des systèmes définis.

Pour aboutir à ce résultat, les concepts d'*ACME* représentent une intersection des concepts communs aux ADLs qui furent ses prédécesseurs. Comme noyau, *ACME* définit une ontologie pour les architectures, regroupant sept notions. Comme dans nombre d'ADLs un composant représente une unité de traitement ou contenant des données d'une application. Il est défini par une interface qui, comme dans le contexte de *Wright*, est composée de un ou plusieurs ports. Les connecteurs représentent les interactions possibles entre types de composants. Leurs interfaces sont définies par un ensemble de types de rôles, chacun spécifiant un participant à l'interaction. La notion de configuration que nous avons défini au début de ce chapitre s'appelle système dans le contexte d'*ACME*. Les notions de représentation et de carte de représentation permettent de décrire hiérarchiquement une application, la représentation définissant un sous-élément raffinable.

En plus de ces concepts, *ACME* offre un mécanisme d'annotation pour définir des propriétés non liées à la structure, et support de l'intégration et de l'extensibilité. Enfin, *ACME* offre la possibilité de définir des patrons de description (*templates*) pour réutiliser tout ou partie des spécifications (similaire aux styles). Les ADLs visent un grand nombre d'applications dans des contextes totalement différents. Il n'est donc pas possible de fournir des propriétés configurables sur les composants qui visent tous les cas possibles. Le mécanisme d'annotation d'*ACME* répond à ce besoin en permettant la définition de propriétés, comme des contraintes de temps réel ou de qualité de service, dans les types de composants et de connecteurs. Les *propriétés* servent aussi à décrire des propriétés provenant d'ADLs intégrés dans un système défini avec *ACME*.

3.3.1.2 Mise en œuvre

La figure 3.3 présente une version simplifiée de la définition du dîner des philosophes avec *ACME*. Le système *Diner* définit l'architecture de l'application. Ce système regroupe la définition de deux composants, *philososphe* et *fourchette*, ainsi que d'un connecteur *main*. Les ports de chaque composant sont précisés, *gauche* et *droite* pour les mains du philosophe, et *prendre* pour saisir une fourchette. Le composant fourchette dispose d'un état *libre* précisant

si il est disponible ou non. Le connecteur définit un ensemble de rôles et par quels composants ces rôles sont joués au travers de la clause *attachements*. Enfin, le connecteur précise une propriété non fonctionnelle : le fonctionnement en mode synchrone.

```
System Diner = {
  Component philosophe = {
    Port gauche ;
    Port droite ;
  }
  Component fourchette = {
    Port prendre ;
    Properties = {
      libre : boolean = true ;
    }
  }
  Connector main = {
    Roles {philo, fork}
    Properties = {
      synchronous : boolean = true ;
    }
    Attachements = {
      philosophe.gauche to main.philo ;
      fourchette.prendre to main.fork ;
    }
  }
}
```

FIG. 3.3 – *Exemple de définition d'architecture avec* ACME *(extrait)*

3.3.1.3 Evaluation

ACME propose un support permettant à la fois d'**intégrer des architectures** décrites potientiellement à l'aide d'ADLs variés, et de **définir de nouveaux ADLs**. Le cœur d'*ACME* vise principalement à la **définition de la partie structurelle d'une application**, sans pour autant permettre l'expression de la dynamique d'une architecture.

ACME se limite donc à un formalisme (puisqu'il n'y a pas de génération de code possible à partir des descriptions) assez faible en terme de sémantique (puisque reposant sur l'utilisation de formalismes et d'outils externes) et sans séparation des préoccupations (par exemple les propriétés fonctionnelles et non fonctionnelles sont mélangées au sein des annotations).

ACME peut donc représenter un support du rôle d'architecte et de la communication architectes / développeurs, mais toute utilisation autre passera par la définition d'une ontologie pour les annotations relatives aux autres acteurs. *ACME* est donc intéressant en terme d'extensibilité du langage, mais impose pour chaque domaine d'activité de produire, de façon *ad hoc*, les ontologies des préoccupations et les outils associés à leur exploitation.

3.3.2 xArch et xADL 2.0

3.3.2.1 Modèle

xArch a été développé en commun par l'institut de recherche en logiciel de l'université de Californie à Irvine et l'université de Carnegie Mellon à Pittsburg. Il représente un effort pour faire évoluer des ADLs et outils propriétaires vers une représentation plus ouverte et flexible des architectures logicielles. Le cœur d'*xArch* comprend les concepts communs à la grande majorité des ADLs sans poser de contrainte sur leur utilisation en termes de comportement et d'agencement. Pour les autres préoccupations, les concepteurs peuvent ajouter autant de concepts que nécessaires à cette base de manière indépendante et incrémentale. De par son approche, il est possible de réutiliser avec *xArch* des éléments d'architectures logicielles existantes, faisant de *xArch* une alternative à *ACME* pour l'échange de définitions d'architectures.

La spécification *instance*, le cœur d'*xArch*, fournit un support à la définition d'architectures découpé en trois éléments. Les *instances architecturales* permettent de modéliser la structure d'exécution d'un système en termes d'instances de composants, connecteurs, interfaces et liens. Ces derniers mettent en relation les composants et les connecteurs. Ainsi, les types de connecteurs qui sont explicites sont plus aisément réutilisables. La notion de *groupe au sens général* offre un mécanisme pour regrouper des instances d'éléments de manière arbitraire, *i.e.* selon les besoins des concepteurs. Enfin, la notion de *sous-architecture* permet de définir des instances de composants et de connecteurs ayant une architecture interne, elle aussi exprimée à l'aide de *xArch*.

xADL 2.0 [21, 22, 20] est un format de représentation d'architectures défini au *Institute for Software Research* de l'université de Californie à Irvine. Le contexte de ce travail est la volonté de représenter les architectures de familles de produits. Une architecture de famille de produits est perçue comme une architecture logicielle *normale* contenant plusieurs points de variation bien définis. *xADL 2.0* a donc été développé sur les constats suivants. Les architectures de familles de produits ont un ensemble de concepts en commun dans leurs représentations. Les représentations d'architectures de familles de produits sont des représentations d'architectures enrichies de fonctionnalités qui ont pour but de capturer les aspects liés à la notion de famille de produits. *xADL 2.0* utilise le cadre de travail défini par *xArch* pour définir ces représentations.

Les extensions apportées à *xArch* au sein de *xADL 2.0* sont, tout d'abord, d'ordre structurelle et de typage. Au niveau architectural, il est possible de spécifier la vue de conception en supplément de la vue en terme d'instance. Un modèle de typage est fourni relativement au style des langages de programmation. Il permet aux architectes de préciser que certains composants, connecteurs et interfaces partagent un type commun. Les *sous-architectures* apportent la possibilité de spécifier des compositions selon un point de vue conception pour un type de composant ou de connecteur. Ceci est fait de manière similaire aux langages de programmation avec les constructions telles que les structures de données et autres types construits.

Au dessus de ce nouveau noyau d'*xADL 2.0*, d'autres extensions sont définies. L'extension *options* permet de définir des types de composants ou connecteurs comme optionnels dans une architecture. L'extension *variants* permet de définir des types variables, *i.e.* qui se

comportent comme les unions dans les langages de programmation. L'extension *versions* supporte la définition d'arbres de versions des types et leur utilisation concurrente au sein d'une même architecture. L'extension *implantations* étend les spécifications de type pour définir des types abstraits qui seront étendus par les développeurs. Pour terminer, cette liste n'étant pas exhaustive, l'extension *conditions booléennes* propose un schéma pour définir des conditions booléennes, base de la notion de contraintes.

3.3.2.2 Mise en œuvre

La mise en œuvre d'*xArch*, et donc implicitement d'*xADL 2.0*, repose sur le constat que les schémas XML [102, 103, 104] représentent un bon support pour construire des représentations modulaires et extensibles d'architectures. Les concepteurs de logiciels peuvent ajouter des propriétés au langage de représentation de l'architecture pour modéliser des aspects spécifiques à leurs besoins. Le langage XML étant naturellement prolixe, nous ne présenterons ici que l'utilisation du cœur du noyau de *xArch* pour définir un composant et une architecture.

La figure 3.4 présente la définition d'un type de composant (`instance:componentInstance`), d'un type de connecteur (`instance:connectorInstance`) et de liens (`instance:linkInstance`) au sein d'une architecture à l'aide du noyau *xADL*. La description de chaque composant est défini sous forme libre (`instance:description`). Ici nous spécifions le type de composant philosophe. Ensuite, chaque interface du composant (`instance:interfaceInstance`) est décrite de manière libre. Ici nous donnons le nom de l'interface, et la direction, `in` pour l'interface des services offerts par le philosophe et `out` pour l'interface requise par le philosophe (l'interface de connexion vers une fourchette). Nous considérons dans le contexte d'*xArch* que les fourchettes du dîner des philosophes sont des connecteurs. Ce choix est motivé par le fait que les fourchettes ne jouent qu'un rôle « secondaire » ici, elle servent à synchroniser les interactions entre philosophes.

Une fois les types de composants et de connecteurs définis, les liens (`instance:linkInstance`) viennent assembler le tout. Les liens peuvent être vus comme la glue entre les composants et les connecteurs. Pour chaque lien, les deux points de connexion sont définis (`instance:point`). L'un d'entre eux référence une interface de composant et l'autre une interface de connecteur.

Les concepteurs d'*xArch* mettent en avant le fait que nombre de parseurs XML sont disponibles et peuvent servir de base à la production d'outils spécifiques aux besoins des extensions définies par les concepteurs d'ADLs. De plus, les extensions pouvant être hiérarchisées, il est possible d'intégrer l'utilisation d'outils disponibles pour prendre en charge la mise en œuvre de certaines extensions. Par exemple, lors du *parsing* d'une architecture, la prise en charge des aspects liés à l'administration peut reposer sur l'utilisation d'un outil d'administration disponible pour le modèle de composant technologique choisi.

3.3.2.3 Evaluation

xArch représente une **base de formalisme pour définir des ADLs** en rapport avec les besoins des concepteurs. *xArch* est nécessairement à utiliser conjointement avec un modèle de composants. La définition des interfaces, services fournis et requis, des composants et des connecteurs ne se fait pas au niveau de l'ADL mais au niveau du modèle technologique utilisé. Cependant, la définition d'extensions peut permettre de décrire les interfaces directement au

```
<xArch ...>
  <instance:archInstance instance:id="0"
                         xsi:type="instance:ArchInstance">
    <instance:componentInstance instance:id="1"
                                xsi:type="instance:ComponentInstance">
      <instance:description xsi:type="instance:DescriptionInstance">
        name = Philosophe
      </instance:description>
      <instance:interfaceInstance instance:id="2"
                                  xsi:type="instance:InterfaceInstance">
        <instance:description xsi:type="instance:DescriptionInstance">
          name = PhilosopheItf
        </instance:description>
        <instance:direction xsi:type="instance:Direction">
          in
        </instance:direction>
      </instance:interfaceInstance>
      <instance:interfaceInstance instance:id="3"
                                  xsi:type="instance:InterfaceInstance">
        <instance:description xsi:type="instance:DescriptionInstance">
          name = FourchetteItf
        </instance:description>
        <instance:direction ...> out </instance:direction>
      </instance:interfaceInstance>
      <!-- seconde interface de connexion vers une fouchette -->
    </instance:componentInstance>
    <instance:connectorInstance instance:id="4"
                                xsi:type="instance:ConnectorInstance">
      <instance:description xsi:type="instance:DescriptionInstance">
        name = Fourchette
      </instance:description>
      <instance:interfaceInstance instance:id="5"
                                  xsi:type="instance:ConnectorInstance">
        name = FourchetteItf
      </instance:interfaceInstance>
      <instance:direction ...> in </instance:direction>
    </instance:connectorInstance>
    <instance:linkInstance instance:id="6"
                           xsi:type="instance:LinkInstance">
      <instance:point xsi:type="instance:Point">
        <instance:anchorOnInterface xlink:href="#3" ...>
      </instance:point>
      <instance:point xsi:type="instance:Point">
        <instance:anchorOnInterface xlink:href="#5" ...>
      </instance:point>
    </instance:linkInstance>
  </instance:archInstance>
</xArch>
```

FIG. 3.4 – *Exemple de définition d'un type de composant avec* xADL

niveau de l'ADL. Un intérêt de cette approche est la **possibilité d'utiliser une définition *xArch* avec plusieurs modèles de composants** et de choisir le modèle le plus approprié sans que l'ADL ne fige des limites à l'utilisation de ces modèles. Il est par contre requis de définir soit des projections vers les modèles de composants soit de disposer d'outils qui mettent en relation l'ADL et le modèle.

Les capacités d'extensibilités d'*xArch* représentent une réponse au support potentiel de toute préoccupation liée au processus d'ingénierie du logiciel. Mais pour cela, il faut produire les outils de manière *ad hoc*, une fois les extensions définies. De plus, toutes les préoccupations sont mélangées au sein de la description XML. Au travers de la prodution d'un environnement particulier il est possible de rendre disponible des vues sur l'architecture. De plus, un **arbre DOM** (*Document Object Model*) [101] du fichier XML représente une **base pour fournir une réification de l'architecture**.

xArch pourrait donc servir de base à la définition d'un ADL respectant la séparation des préoccupations, mais tout l'environnement associé est à produire. *xArch* représente donc une réponse à la préoccupation d'un architecte et d'un concepteur d'ADL. Il fournit aussi un format d'échange standard entre acteurs, mais l'absence d'outils de base n'en fait pas une solution prête à l'emploi pour répondre aux préoccupations de notre processus d'ingénierie du logiciel.

3.4 Langages de configuration

Cette section présente les deux langages de configuration C2, Olan et les *Component Assembly Descriptors* du CORBA Component Model. L'objectif de ces propositions est d'exploiter les descriptions d'architectures pour produire en partie l'implantation des composants ou de supporter, au moins partiellement, le déploiement des applications.

3.4.1 C2

3.4.1.1 Modèle

C2 [57, 95] est le résultat de travaux menés à l'université de Californie à Irvine (UCI). La préoccupation première de *C2* est de fournir un style architectural reposant sur les composants et les événements, pour les applications comportant des interfaces utilisateurs graphiques. *C2* vise la spécificité de ce type d'applications tout en supportant potentiellement un spectre plus large. Par exemple, les composants peuvent être implantés de manière hétérogène, et répartis, et les architectures peuvent évoluer dynamiquement.

Le style architectural *C2* peut être informellement perçu comme un réseau de composants concurrents reliés par des connecteurs. Chaque composant dispose d'un ou plusieurs flots d'exécution et d'un état. Globalement, une description architecturale en *C2* est organisée selon un modèle vertical. La séparation droite / gauche des services décrite dans la section 2.1 devient une séparation haut / bas en *C2*, aussi bien pour les composants que pour les connecteurs. Un composant *C2* est défini par une interface offrant deux catégories de services : les services requis (le dessus) et les services fournis (le dessous). Un composant est connecté pour ces interfaces à un unique connecteur, mais un connecteur peut être lié à plusieurs composants (aussi bien en dessus que en dessous) ou connecteurs. D'une manière générale,

les liens entre composants et connecteurs se font toujours sur une base dessus vers dessous (deux dessus ou deux dessous ne peuvent être reliés directement).

Le domaine du dessus d'un composant représente l'ensemble des notifications auxquelles ce composant peut répondre ainsi que les événements émissibles vers le haut de l'architecture, *i.e.* les émissions de requêtes. Le domaine du dessous représente l'ensemble des notifications qu'un composant peut produire, ainsi que les requêtes auxquelles il peut répondre. Ceci se justifie par l'approche événementielle induite par le contexte des interfaces graphiques ; aussi bien le système que les usagers peuvent demander des requêtes de façon concurrente et indépendante.

Le style *C2* met en application le principe que la limitation de visibilité est importante dans les styles architecturaux ; elle est ici nommée *substrate independence*. Dans une hiérarchie définissant une architecture, un composant ne peut être conscient que d'autres composants situés au-dessus de lui ; les composants au-dessous lui sont inconnus. L'usager final se trouve au bas de l'architecture alors que le système sous-jacent à l'application est en haut. La limitation de visibilité est mise en avant pour favoriser l'interchangeabilité et la réutilisabilité des composants et des connecteurs. Avec l'intention que les composants ne soient pas fortement dépendants de leur domaine supérieur (les composants dont ils sont conscients), *C2* introduit la notion de traduction d'événements. Les domaines de traduction permettent de transformer la requête d'un composant dans une forme compréhensible par le composant qui reçoit cette requête.

3.4.1.2 Mise en œuvre

La définition d'un type de composant regroupe à la fois son interface et son comportement. Ce type de définition est réalisée à l'aide de la notation de définition d'interfaces (*IDN – Interface Definition Notation*). La figure 3.5 illustre une définition simplifiée pour le type de composant `Philosophe`. L'interface spécifie pour les domaines du dessus et du dessous les événements qui peuvent être émis et reçus par un composant *philosophe*. Dans le cadre du dîner, un composant *philosophe* n'est pas utilisé par d'autres composants, mais il utilise des composants de type *fourchette*, d'où les `null` dans la partie `bottom_domain`. La partie `behavior` précise comment le composant se comporte au démarrage : il pense, puis il essaie d'acquérir les deux fourchettes pour manger. Il ne mangera que s'il reçoit des événements de type `FourchetteAcquise`. La clause `cleanup` précise qu'à sa terminaison, un composant *philosophe* doit libérer les fourchettes. La clause contexte permet de préciser à quel niveau de l'architecture ce composant peut être utilisé. Ici, `top_bottom` est précisé puisqu'un philosophe ne propose pas de service à autrui, il est simplement utilisateur.

Une description d'architecture regroupe, comme dans les autres ADLs présentés, à la fois les types de composants, de connecteurs et les instances utilisées dans l'application. Cette description se fait à l'aide du langage *C2 SADL*, la structure d'une application étant réalisée dans la partie *ADN* (*Architecture Description Notation*). La figure 3.6 présente une version simplifiée de la définition du dîner des philosophes avec le style *C2*. Cette définition utilise celles des composants *Philosophe* présentées précédemment et *Fourchette*. La section `components` présente l'organisation hiérarchique des composants en précisant les deux domaines externes (un domaine intermédiaire peut être défini, mais n'a pas de raison d'être dans notre exemple). En haut se trouvent les fourchettes (composants utilisés) et en bas les philosophes (composants utilisateurs). La clause `component_instances` permet de définir des instances utiles des

```
component Philosophe is
  interface
    top_domain is
      out
        AcquerirFourchettes (f1,f2: NomFourchette) ;
        LibererFourchette (f: NomFourchette) ;
      in
        FourchetteAcquise (f: NomFourchette) ;
    bottom_domain is
      in
        null ;
      out
        null ;
    parameters
      null ;
    methods
      procedure Manger () ;
      procedure Penser () ;
    behavior
      startup
        invoke_methods Penser ;
        always_generate AcquerirFourchettes ;
      cleanup
        always_generate LibererFourchettes ;
      received_messages FourchetteAcquise ;
        invoke_methods Manger ;
  context
    top_bottom philosophe ;
end Philosophe ;
```

FIG. 3.5 – *Exemple de définition d'un type de composant avec* C2

types de composants connus dans l'architecture. Dans le contexte du dîner, un seul connecteur est défini pour toutes les instances. Ceci est dû au fait qu'un seul connecteur peut être lié au dessus et un seul au dessous d'un composant. Ce connecteur va filtrer les messages en fonction des noms des composants, pour que seul le destinataire d'un message le reçoive, ceci sera implémenté sous forme de condition par rapport au nom des composants. Enfin, la dernière clause précise la topologie de notre application, *i.e.* comment les composants et le connecteur sont liés les uns aux autres.

```
architecture Diner is
  components
    top_most
      Fourchette ;
    bottom_most
      Philosophe ;
  component_instances
    philosophe1 instantiates Philosophe ;
    fourchette1 instantiates Fourchette ;
    ...
  connectors
    connector Main is
      message_filter conditional ;
    end Main1gauche ;
  architectural_topology
    connector Main connections
      top_ports
        philosophe1 ;
        philosophe2 ;
      bottom_ports
        fourchette1 ;
        fourchette2 ;
end Diner ;
```

FIG. 3.6 – *Exemple de définition d'architecture avec le style* C2

L'environnement *C2* permet aussi d'agir sur une architecture pour en modifier sa structure dynamiquement. Il est possible de créer, détruire des composants, des connecteurs et les liens entre eux. Ces opérations sont accessibles au cours de l'exécution d'une application, mais ne sont pas directement exprimées dans la définition d'une architecture. Un exemple de modification dynamique est illustrée dans la figure 3.7. Les opérations sont exprimées à l'aide du langage *ACN* (*Architecture Construction Notation*) et présentent grossièrement l'ajout d'un composant philosophe et d'un composant fourchette. Les deux premières opérations ajoutent les composants et les deux dernières relient ces composants avec le connecteur *Main*.

Dans le contexte de *C2*, un composant est constitué d'un ensemble d'objets l'implantant et regroupés au sein d'un *wrapper*. L'intérêt de cette approche est de pouvoir, d'une part, offrir une vision et une utilisation homogène de composants hétérogènes et, d'autre part, de pouvoir réutiliser des implantations de composants existants. Un composant se décompose en deux parties : le code fonctionnel qui est regroupé au sein du *wrapper*, et le code gérant la composition et les contraintes imposées aux composants, résultante de la partie *behavior* de

```
Diner.add (Philosophe) ;
Diner.add (Fourchette) ;
Diner.weld (Main, Philosophe) ;
Diner.weld (Fourchette, Main) ;
```

FIG. 3.7 – *Exemple de modification dynamique d'architecture avec le style* C2

sa définition. Entre ce dernier module et les connecteurs, les requêtes peuvent transiter par les traducteurs de domaine. Le cadre d'intégration de l'implantation est défini dans l'environnement *C2* qui permet de générer la partie *wrapper*, et la partie liée à la composition des implantations de composants. Pour exprimer la relation entre partie générée et implantation écrite manuellement, c'est ici aussi la notation *ADN* qui est utilisée, cette fois-ci avec les déclarations de type *system*. Cette notation est brièvement présentée dans la figure 3.8. Les clauses *is_bound_to* associent les types de composants avec leur implantation. Parallèlement, les connecteurs sont générés à partir de leur définition de la clause *architecture*.

```
system DinerImpl is
  architecture Diner is
    Philosophe is_bound_to PhilosopheImpl ;
    Fourchette is_bound_to FourchetteImpl ;
end DinerImpl ;
```

FIG. 3.8 – *Exemple de relation architecture / implantation avec le style* C2

3.4.1.3 Evaluation

Le style architectural *C2* représente une première réponse intéressante aux préoccupations présentées dans notre processus d'ingénierie du logiciel. Bien qu'il ne soit pas possible de spécifier les propriétés non fonctionnelles et que la sémantique de *C2* soit figée, un architecte dispose de moyens intéressants pour **spécifier l'architecture d'une application** et, jusqu'à un certain point, **son dynamisme**. Les développeurs disposent, quant à eux, d'un **cadre pour réaliser les implantations fonctionnelles**. Bien que disposant du langage *ADN* pour mettre en relation des types et des implantations, le rôle des intégrateurs n'est pas totalement adressé. Les administrateurs disposent avec l'expression de la dynamique d'une amorce de moyens pour administrer les applications.

Toutefois, l'aspect déploiement des applications n'est pas du tout pris en compte. Ni la répartition logique, ni la projection vers une répartition physique des composants ne peuvent être décrites dans le contexte de *C2*. Il est évident que l'objectif premier de cet ADL était la définition des interfaces graphiques, donc principalement dans le contexte des applications monolithiques, la répartition devient secondaire pour ses auteurs. Il en va de même dans ce contexte pour la représentation à l'exécution de la description de l'architecture qui se justifie moins dans ce contexte.

3.4.2 Olan

Olan [6, 7, 8], développé au laboratoire SIRAC de l'INRIA Rhône-Alpes, a pour objectif d'offrir un langage de configuration pour construire, déployer et jusqu'à un certain point administrer des applications réparties. Tout comme Darwin [47], Olan est un environnement de configuration plus riches que les ADLs.

3.4.2.1 Modèle

L'environnement Olan est destiné à configurer des applications réparties. Le terme configuration désigne d'une part l'action de définition et de spécification de l'architecture de l'application et de son mode d'exécution, et d'autre part l'action d'installer, de déployer et d'exécuter l'application répartie sur le système informatique qui l'héberge.

Les éléments centraux du modèle Olan sont les composants et les connecteurs. Deux types de composants sont définis : les *composants primitifs*, unités utilisées pour l'intégration du logiciel, et les *composants composites*, unités de structuration.

- Le composant primitif est l'unité d'encapsulation de logiciel existant. La façon dont le code encapsulé est lié au composant ainsi que sa nature (code source, classes, bibliothèque, exécutable) et son langage de programmation sont décrits.
- Les composants composites sont l'unité de structuration. Ils servent à la fois d'entités de description de la configuration et d'entités de structuration d'une application en modules ou composants coopérants. Les composites permettent de former une hiérarchie de composants, hiérarchie partiellement ou totalement réutilisable dans diverses applications.

Les connecteurs sont utilisés au sein des composants composites pour spécifier le protocole de communication entre les sous-composants. Une propriété importante des connecteurs est de permettre l'interconnexion de composants hétérogènes, d'adapter le flot de données et de mettre en oeuvre le schéma de coordination souhaité en fonction des interfaces interconnectées.

Olan s'est vivement intéressé à la répartition des composants lors du déploiement en étudiant le problème du placement des différents composants d'une application dans des processus, répartis sur un ensemble de sites. La base de la proposition consiste à spécifier pour chaque composant le site mais aussi l'utilisateur pour lequel ce composant doit s'exécuter. De plus, une approche intéressante du déploiement est d'associer des critères de placement plutôt qu'une désignation directe d'un nom de site ou d'utilisateur.

Ces expressions expriment les règles d'administration comme des contraintes liant les noeuds d'exécution et les utilisateurs pour lesquels les composants doivent être exécutés. Cette description est indépendante de la mise en oeuvre des composants afin de pouvoir exprimer différentes règles de configuration pour des composants identiques. Les travaux menés autour de la plate-forme à agents AAA (*Agent Anytime Anywhere*) [23, 80] sont un prolongement de ces travaux sur la configuration et la reconfiguration des applications Olan.

3.4.2.2 Mise en œuvre du modèle

Pour la mise en oeuvre de ce modèle, Olan propose un langage de configuration d'application appelé OCL (Olan Configuration Language) qui est un langage de description d'archi-

```
component Philosophe {
    interface PhilosopheItf ;
    implementation <implementation_identifier> ;
    management <management_identifier> ;
}
```

FIG. 3.9 – *Description d'un composant avec Olan*

tecture logicielle. Celui-ci a pour rôle de décrire l'application en définissant les composants, les connecteurs nécessaires à l'application et ensuite l'interconnexion entre les composants. La figure 3.9 présente un synopsis de définition de composants à l'aide de ce langage. La figure 3.10 illustre la représentation graphique inspirée de Darwin.

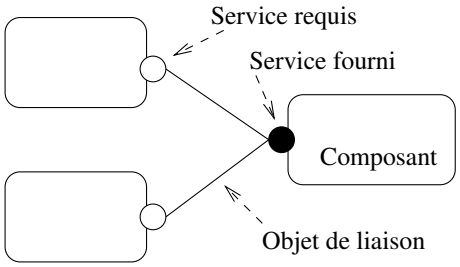


FIG. 3.10 – *Représentation graphique de composants avec Olan*

Dans le contexte d'*Olan*, une configuration est spécifiée sous forme de composants composites. Les composites regroupent à la fois les composants et leur interconnexion. La définition des composites suit le synopsis présenté dans la figure 3.9. La clause `management` regroupe la définition de la composition.

```
Component Diner {
  interface DinerItf ;
  implementation ... ;
  management ... ;
}
```

FIG. 3.11 – *Composant composite dans Olan*

La spécification des paramètres d'administration repose sur un ensemble de conditions qui doivent être satisfaites par le contexte d'exécution du composant. Pour cette raison, chaque composant possède deux attributs d'administration, structurés en différents champs (voir figure 3.12). Le premier caractérise le noeud d'exécution, et le second l'utilisateur qui exécute le composant. Les conditions sont des expressions booléennes, chacune d'elles exprimant une contrainte pour le déploiement d'un composant particulier.

```
Management attribute Host {
    string name ;     // nom du noeud
    string IPAdr ;    // Adr IP
    string platform ; // Architecture
    string os ;       // type d'OS (POSIX, nt, etc.)
    short osVersion ; // Numéro de version de l'OS
    long CPULoad ;    // charge moyenne pour les 10 dernières mn.
    long UserLoad ;   // nombre de connexions
}
Management attribute User {
    // caractérisation du propriétaire du composant
    string name ;            // user name
    long UID ;               // user ID
    sequence <long> grpId ;  // group ID
}
```

FIG. 3.12 – *Attributs de déploiement dans Olan*

3.4.2.3 Evaluation

Olan, contrairement aux autres ADLs présentés précédemment, tient compte de la conception, du développement et de l'assemblage de composants. Il propose aussi un algorithme de déploiement basé sur la définition d'attributs de configuration. L'approche Olan présente les avantages suivants :

- le langage permet de **décrire de manière hiérarchique les applications** ;
- le langage de description a été étendu vers un langage de configuration, permettant de **rendre explicite la structure dynamique des applications** [80] ;
- le langage intègre les **aspects liés à l'administration** des applications.

Cependant ce modèle, comme les projets précédents, propose une solution propriétaire, non basée sur les langages de type IDL émergeants. De plus, la proposition de déploiement effectuée est une base qu'il faut poursuivre afin d'offrir plus de souplesse. En effet, les attributs sont fixés et bornés.

3.4.3 Descripteurs d'assemblages du CCM

3.4.3.1 Modèle

Les descripteurs d'assemblage de composants (CAD, *Component Assembly Descriptor*) du modèle de composants CORBA [69] représentent la proposition de l'OMG (*Object Management Group*) pour décrire des applications à base de composants. Ils font partie d'un ensemble de descripteurs destinés à documenter un composant, une archive de composants et un assemblage. L'objectif principal de ces descripteurs d'assemblages est de pouvoir diffuser et déployer de manière automatique tout ou partie d'une application. Cette section vient en complément de la présentation du modèle de composants CORBA (voir section 2.4.2).

Le descripteur d'assemblage offre un patron pour instancier un ensemble de composants et pour les interconnecter. Il décrit l'assemblage des composants, c'est-à-dire les éléments de

description des composants, des connexions et du partitionnement (placement). Les instances de composants sont connectées au travers des déclarations *provides* / *uses* et *emits* / *consumes*. Le descripteur référence les fichiers de chaque composant et décrit les ports mis en jeu dans chaque connexion. Ce descripteur définit les regroupements logiques des composants qui seront projetés au déploiement sur des sites physiques.

La description d'un type de composant est réalisée à l'aide de *descripteurs de composants CORBA* (CCD, *CORBA Component Descriptor*). Ce descripteur précise les caractéristiques d'un composant spécifiées pendant les phases de conception et de développement. Il décrit la structure d'un composant : relations d'héritage, interfaces supportées, ports, *etc.* Ce descripteur précise aussi les contraintes techniques liées au composant et les propriétés non fonctionnelles de celui-ci. Enfin, le *descripteur de propriétés* (CPF, *Component Property File*) définit l'état initial d'un composant. Il précise les valeurs des attributs à fixer à la création d'une instance de composant.

3.4.3.2 Mise en œuvre

Les descripteurs du modèle de composants CORBA sont basés sur une évolution du langage OSD (*Open Software Descriptor*) du W3C (*World Wide Web Consortium*). Ce langage est défini comme un vocabulaire XML et regroupe quatre DTD (*Data Type Definition*). L'OMG a étendu ce langage pour prendre en compte les spécifités des composants CORBA.

La figure 3.13 présente un extrait de la définition du dîner des philosophes avec un descripteur d'assemblage du CCM. La première partie (`<componentfiles>`) précise les archives de composants contenant les implantations à utiliser pour déployer l'application. Un identifiant est affecté à chaque archive pour la référencer dans la suite du descripteur.

La seconde partie (`<partitionning>`) précise le découpage logique de l'application, le placement des composants. Pour chaque instance de composants, sa fabrique (`<homeplacement>`) et l'archive le contenant (`<componentfileref>`) sont précisés. Enfin, un identifiant est affecté à chaque instance (`<componentinstantiation>`).

La dernière partie de ce descripteur (`<connection>`) stipule comment les instances de composants sont interconnectées. Ici une connexion synchrone (`<connect>`) est définie entre l'instance de composants *philosophe* et l'instance de composant *fourchette*. Les ports mis en jeu dans cette connexion sont spécifiés. La facette (`<providesport>`) fournie par la fourchette est identifiée (`<providesidentifier>`) pour l'instance dont l'identifiant est `Bb`. Le réceptacle correspondant (`usesport`) est ensuite précisé : le port `main_gauche` de l'instance de composant philosophe `Aa` est utilisé (`usesidentifier`).

3.4.3.3 Evaluation

Les descripteurs d'assemblage du modèle de composants CORBA (CCM) représentent une forme minimale de l'expression d'une architecture. Ils ne sont pas aussi complets que les autres langages de description d'architectures présentés dans ce chapitre. Les préoccupations majeures prises en compte dans les descripteurs d'assemblage du CCM sont :

- la description de la **structure de l'application** ;
- la définition du **placement des instances de composants** ;
- la définition de **l'intégration des implantations** de composants à utiliser ;

```
<componentassembly id="XYZ:0987654321">
  <description>
    Exemple de descripteur d'assemblage de composants:
    le dîner des philosophes.
  </description>

  <componentfiles>
    <componentfile id="A">
      <fileinarchive name="philosophe.csd" />
    </componentfile>
    <componentfile id="B">
      <fileinarchive name="fourchette.csd" />
    </componentfile>
  </componentfiles>

  <partitionning>
    <homeplacement id="AaHome">
      <componentfileref idref="A" />
      <componentinstanciation id="Aa" />
    </homeplacement>
    <homeplacement id="BbHome">
      <componentfileref idref="B" />
      <componentinstanciation id="Bb" />
    </homeplacement>
    <componentplacement>
  </partitionning>

  <connection>
    <connect...>
      <providesport>
        <providesidentifier>manche</providessidentifier>
        <componentinstanciationref idref="Bb">
      </providessport>
      <usesport>
        <usesidentifier>main_gauche</usesidentifier>
        <componentinstanciationref idref="Aa">
      </usesport>
    </connect...>
   </connection>
  </componentassembly>
```

FIG. 3.13 – *Exemple d'assemblage de composants pour le CCM (extrait)*

- la description des **propriétés non fonctionnelles** des composants d'une architecture ;
- l'utilisation de la description de l'architecture comme **support du déploiement des applications** ;
- la possibilité d'**échanger la description d'une architecture** entre des outils de fournisseurs divers.

En contrepartie, les connexions entre instances de composants définies au sein d'une architecture ne sont pas réifiées sous forme de connecteurs à l'exécution. Il n'est pas non plus possible d'organiser les interconnexions de composants de manière hiérarchique, par exemple au travers de composites. Ces deux limitations sont relatives à la définition du modèle de composants CORBA. Enfin, le langage CAD de l'OMG, bien que défini en XML, n'est pas prévu pour être étendu. Il est néanmoins possible de l'étendre, mais le résultat devient alors une solution propriétaire pour laquelle des outils particuliers doivent être produits.

3.5 Conclusion

Les différents langages de description d'architectures (ADLs) présentés dans ce chapitre prennent en compte un certain nombre de préoccupations. Nous résumons ici les différentes préoccupations rencontrées dans cette présentation.

Structure La structuration est la préoccupation minimale d'un langage de description des architectures : fournir une représentation de l'architecture.

Validation La validation sert à garantir qu'une interconnexion de composants peut fonctionner.

Simulation La simulation permet de faire une évaluation du comportement d'une application en observant les interactions entre les différents composants.

Dynamique La définition de la dynamique d'une application permet de préciser ses évolution possibles (et anticipées). Par exemple, l'ajout ou le remplacement d'un certain nombre de composants et de connexions à l'exécution.

Génération La génération automatique permet de produire, au moins en partie, l'implantation des composants logiciels identifiés comme faisant partie de l'architecture.

Intégration L'intégration consiste à préciser les implantations de composants à utiliser pour créer une instance de l'application définie par une architecture.

Placement Le placement consiste à préciser la localisation des instances de composants d'une application dans un environnement d'exécution.

Déploiement La prise en charge de cette préoccupation signifie que le processus de déploiement d'une application peut être, au moins en partie, automatisé à partir de la description de son architecture.

Administration Les moyens d'administration peuvent être fournis à partir de la définition d'une architecture et de sa dynamique.

Echange L'échange caractérise la possibilité d'utiliser un ADL pour partager la description d'architectures entre environnements ou utilisateurs ne collaborant pas au sein d'un processus logiciel.

Extensibilité L'extensibilité est la capacité d'un ADL à permettre l'ajout de nouveaux concepts non prévus initialement.

Les concepts disponibles dans les ADLs sont eux aussi variables. Nous résumons ici les différents concepts que nous avons rencontrés dans ce chapitre.

Composant Brique de base des applications, le composant est l'élément minimal de tout ADL (présenté dans ce chapitre).

Connecteur La mention de ce concept précise que les connecteurs existent réellement dans les applications, *i.e.* que l'on peut les manipuler à l'exécution.

Propriétés Les propriétés non fonctionnelles regroupent les aspects d'un composant logiciel dont la prise en charge peut être automatisée. Idéalement, elle sont décrites et non programmées.

Hiérarchique Ce concept signifie la capacité d'un ADL à organiser les interconnexions de composants. Cette notion est souvent réifiée par les composites.

Style Les styles architecturaux sont des patrons architecturaux qui peuvent être définis et utilisés dans différentes descriptions d'architectures.

Les préoccupations de chaque ADL sont différentes. Comme pour les modèles de composants, utiliser un ADL revient donc à faire un choix qui doit être guidé par les besoins d'un processus logiciel. Le tableau de la figure 3.14 présente les préoccupations et la disponibilité des concepts pour chaque ADL présenté ici. La présence d'un « x » signifie la disponiblité, un « - » signifiant que c'est disponible dans une moindre mesure[1].

	Rapide	Wright	ACME	xArch	CAD	C2	Olan
Structure	x	x	x	x	x	x	x
Validation	x	x					
Simulation	x	x					
Dynamique						x	x
Integration			x	x	x		x
Placement					x		x
Génération						x	x
Déploiement					x		x
Administration							x
Echange			x	x	x		
Extensibilité			x	x			
Composant	x	x	x	x	x	x	x
Connecteur	-	x	x	x	-	x	x
Propriétés			x		x	x	x
Hiérarchique		x					x
Style		x	x	x		x	
Réification				-			-

FIG. 3.14 – *Synthèse des préoccupations et concepts présents dans chaque ADL*

It is recognized that architecture should have a strong influence over the life cycle

1. La réification des architectures n'est pas disponible directement dans le cadre d'*xArch*, mais peut être mise en œuvre au travers d'une version DOM du document XML.

> *of a system. [...] However, the concepts of architecture are not yet consistently defined and applied over the life cycle.* [27]

Cette affirmation est bien sûr intimement liée à la définition que l'on fait du cycle de vie d'une application. Pour positionner celle-ci dans le cadre de notre travail, nous avons défini le cycle de vie comme débutant à la phase de conception, et se poursuivant jusqu'à l'exécution des applications. En cela, l'affirmation est aujourd'hui totalement correcte et nous visons à lui proposer une réponse, c'est-à-dire à **fournir un environnement où l'architecture devient réellement centrale et cohérente tout au long d'un processus d'ingénierie du logiciel**. Nous attendons donc d'un ADL la possibilité de réifier à l'exécution les architectures définies, ce que n'offrent pas réellement les ADLs actuels.

L'ambition de définir un ADL généraliste et complet est certainement louable, mais ne nous semble pas une option intéressante. En effet, quels doivent être alors les concepts à prendre en considération ? Si l'on souhaite répondre à toutes les préoccupations présentées dans ce chapitre, cet ADL « ultime » doit donc être une union de tout ce que nous venons de voir, mais ceci soulève plusieurs problèmes. Premièrement, même si l'ADL ainsi défini arrive à être complet, il ne serait complet qu'« aujourd'hui ». L'apparition de nouveaux besoins dans le futur remettra totalement sa complétude[2] en cause. Ensuite, cet ADL deviendra encore plus complexe à utiliser que les ADLs que nous avons présenté ici. Ce n'est donc pas un réel progrès, l'intérêt de disposer d'une solution « complète » mais inutilisable étant bien insignifiant. Enfin, les définitions d'architectures réalisées avec un tel ADL devraient être partagées par tous les acteurs ; mais se pose alors la question de « qui fait quoi ». Nous pouvons donc raisonnablement conclure que cette approche, même si l'ADL était extensible, n'est pas à considérer.

Nous en sommes ainsi arrivés au constat suivant. **Les langages de description d'architectures sont complémentaires aux modèles de composants, par contre leurs modèles, eux, ne sont pas nécessairement complémentaires**. Aujourd'hui, le choix d'un modèle de composants implique un faible pouvoir d'expression des architectures. En contrepartie, le choix d'un langage de description d'architectures implique un modèle de composants particulier et limité. Il est donc important de **considérer une approche qui tire parti de ces deux mondes pour en bénéficier au maximum**. Mais, il ne faut pas non plus tomber dans le travers d'une réponse liant fortement le modèle de composants et le langage de description d'architectures. Dans ce cas, ils ne pourraient évoluer indépendamment et l'on serait toujours limité par le plus faible des deux.

Pour arriver à ce résultat, il est important de disposer de moyens permettant de définir et d'exploiter un ADL, puis à partir de cet ADL, de construire des applications réparties à base de composants et de produire les environnements associés. Enfin, **il est important de structurer la définition et l'utilisation d'un ADL**.

Au-delà de leurs syntaxes, les ADLs présentés dans ce chapitre ont un modèle, toutefois non explicite et figé. Les techniques de méta-modélisation, telles que celles proposées par l'*Object Mangement Group*, représentent une solution pour définir le modèle d'un ADL. L'ap-

2. La complétude n'est pas à comprendre au sens mathématique, mais plutôt au sens couteau suisse : qui offre toutes les fonctionnalités utiles à son utilisateur.

proche dirigée par les modèles (*Model Driven Architecture*) représente quant à elle une manière de structurer la définition de modèles. Le chapitre suivant présente ces deux propositions.

Chapitre 4

Méta-modélisation, la vision de l'*Object Management Group*

Les intergiciels (*middleware*) comme CORBA [71], CCM [69], J2EE [90], EJBs [24] et .Net [97] proposent des abstractions pour définir des applications indépendamment des plates-formes ou langages à utiliser lors de leur mise en œuvre. Toutefois, il n'existe pas d'intergiciel universel et leur multiplication entraîne le besoin croissant d'un niveau abstraction supérieur. Ce dernier doit permettre de capitaliser la définition des applications et les rendre plus perennes et portables au dessus de ces différentes technologies « du milieu ».

L'utilisation de modèles de définition des applications permet d'atteindre ce niveau d'abstraction. Les modèles d'applications et de systèmes permettent de s'abstraire totalement des considérations techniques comme les plates-formes et les langages de programmation. Il est ainsi possible de capitaliser la définition d'une application afin de la mettre en œuvre dans le cadre de différentes technologies.

L'écriture de programmes requiert l'utilisation de langages de programmation. Parallèlement, la définition de modèles requiert la mise en œuvre de *méta-modèles* [1], c'est-à-dire l'ensemble des concepts et leurs relations à utiliser pour exprimer des modèles. Ces techniques sont donc intéressante pour la définition de modèles d'ADLs.

Ce chapitre présentent la vision de l'*Object Management Group* vis-à-vis des techniques de méta-modélisation. Cette présentation vise à donner une vue d'ensemble sur l'intérêt et les moyens associés au domaine de la méta-modélisation. Elle résume aussi brièvement l'intérêt de la séparation des préoccupations. Sa mise en œuvre est discutée dans le cadre de la proposition de l'OMG, visant à diriger les architectures par les modèles.

- La section 4.1 présente l'approche et l'objectif visé par les techniques de méta-modélisation.
- La section 4.2 présente le *Meta Object Facility* (MOF). Cette proposition de l'*Object Management Group* (OMG) fournit un cadre de travail pour définir des méta-modèles.
- La section 4.3 présente l'approche par séparation des préoccupations et donne deux exemples de mise en œuvre.
- La section 4.4 présente le nouveau cadre structurant de mise en œuvre des techniques de modélisation et de méta-modélisation au sein de l'OMG : la *Model Driven Architecture*.

Enfin, la section 4.5 résume l'approche par méta-modélisation et met en avant ses principaux avantages et inconvénients.

1. **méta**: du grec, signifiant *à propos de*.

4.1 Objectifs de la méta-modelisation

4.1.1 Définitions

Afin de préciser les deux termes majeurs de ce chapitre, nous considérons les définitions suivantes [14] :

- Un **modèle** est une abstraction d'un système qui devrait être plus simple que celui-ci. Cette simplification se traduit, en général, par la disparition de détails d'ordre technique. Un modèle représente le système qu'il décrit et doit pouvoir être utilisé à sa place pour répondre à un certain nombre de questions sur celui-ci.
- Un **méta-modèle** est la spécification d'une abstraction, *i.e.* d'un ou plusieurs modèles. Cette spécification définit un ensemble de concepts importants pour exprimer des modèles, ainsi que les relations entre ces concepts. Le méta-modèle définit la terminologie à utiliser pour définir des modèles.

4.1.2 De la nécessité de méta-modèles

La modélisation est une technique de conception de systèmes (par exemple) utilisant un certain nombre de concepts prédéfinis. La méta-modélisation est une technique de définition des concepts à utiliser pour modéliser des systèmes. La méta-modélisation apporte donc la flexibilité nécessaire à la fourniture de moyens adaptés aux besoins d'un processus logiciel, pour concevoir des applications. Deux expressions clés [61] résumant cette approche sont :

- « une tentative pour décrire le monde » et
- « dans un objectif particulier ».

La seconde exprime le fait qu'il ne peut pas y avoir *un* méta-modèle universel utilisable pour décrire tous les systèmes informatisés, du programmateur de la machine à laver au réseau mondial de contrôle du traffic aérien. Il est donc important de comprendre qu'un (méta-)modèle doit être défini pour un objectif précis. Et donc, qu'il existe une multitude de (méta-)modèles. On retrouve ici le besoin de disposer de moyens spécialisés pour être en adéquation avec les besoins d'un domaine donné.

Le dernier aspect d'un méta-modèle, en plus des concepts et des relations du domaine d'application, est la sémantique. Il est en effet important de donner un sens aux éléments définis dans un méta-modèle. Il est encore plus important de partager ce sens entre ses différents utilisateurs. L'utilisation d'assertions logiques ou d'un moyen de description formel [86, 15] sont des possibilités permettant d'introduire de la sémantique dans un méta-modèle.

4.1.3 Mise en œuvre des techniques de méta-modélisation

Les techniques de méta-modélisation sont mises en œuvre dans différents contextes.

Ingénierie des connaissances Des travaux ont porté sur l'utilisation de méta-modèles comme base de la représentation et de gestion des connaissances [33, 35, 92]. Un méta-modèle est alors un moyen d'exprimer l'ontologie d'un domaine.

Production d'applications Cette utilisation des techniques de méta-modélisations est la plus répandue dans les approches « objet » et « composants » de la communauté du génie

logiciel. Elle peut servir de support à la génération de code ou encore à la supervision des systèmes [88].

Workflow Les processus peuvant être modélisés, les techniques de méta-modélisation sont aussi utiles dans les activités liées au *workflow* [78].

4.2 Le Meta Object Facility

Le *Meta Object Facility* (MOF) [18, 65] représente le cœur des aspects de méta-modélisation traités à l'OMG. UML (*Unified Modeling Language*) [73], par exemple, est définissable comme un méta-modèle reposant sur les concepts du MOF. Dans nos travaux, nous avons fait le choix de travailler directement avec le MOF. Cette section présente ce qu'est et ce qu'offre le MOF pour mettre en œuvre les techniques de méta-modélisation.

4.2.1 Présentation

4.2.1.1 Méta, méta, méta... quand tu nous tiens !

Le *Meta Object Facility* (MOF) est la technologie proposée par l'OMG pour définir des méta-données et les représenter par des objets CORBA. Dans ce contexte, une méta-donnée est un terme générique attribué à une donnée représentant de l'information. Une méta-donnée peut aussi bien représenter de l'information contenue dans un système que le système lui-même. De plus, cette description peut se faire à tout niveau d'abstraction requis. Dans le contexte du MOF, un modèle est une collection de méta-données mises en relation des manières suivantes.

- Les méta-données décrivent des éléments d'information, eux même en relation.
- Toutes les méta-données respectent les mêmes règles de structuration et de cohérence, la syntaxe abstraite commune.
- Enfin, les méta-données ont un sens au sein d'un cadre (*framework*) sémantique commun.

Le but du MOF est de fournir un cadre de travail (*framework*) supportant tout type de méta-donnée et permettant la définition de nouveaux types au fur et à mesure de l'évolution des besoins. Pour atteindre cet objectif, le MOF repose sur une décomposition de son architecture en quatre niveaux, ce qui respecte l'approche classique dans les communautés telles que l'ISO et l'EIA (Electronic Industries Association), où cette approche a été mise en œuvre, par exemple, pour la définition du format d'échange standard CIDF (CASE Data Interchange Format) [28]. La clé de voute de cette architecture est la présence d'un niveau de méta-méta-modélisation fournissant un langage commun, permettant de définir et de lier ensemble méta-modèles et modèles. Le modèle MOF correspond au méta-méta-modèle, c'est-à-dire au quatrième niveau de ce cadre. Ce modèle MOF est utilisé pour définir la structure et la sémantique de méta-modèles.

4.2.1.2 L'architecture à quatre niveaux du MOF

Le cadre classique de méta-modélisation repose sur une l'architecture illustrée par la figure 4.1 et définie comme suit. Pour illustrer notre propos, nous ferons un parallèle avec les langages

orientés objets à classes, et plus précisément avec l'exemple classique du dîner des philosophes.

M0 est le niveau contenant les informations à décrire. Pour faire le parallèle avec les langages objets, le niveau M0 contient les instances de classes, par exemple les instances de philosophes et de fourchettes.

M1 est le niveau contenant les méta-données décrivant l'information et regroupées sous forme de modèles. Toujours dans le cadre de notre parallèle, ce niveau contient la définition des classes, ce que sont un philosophe et une fourchette. De manière simplifiée, un philosophe sera défini à l'aide de deux attributs, un nom et une association vers deux fourchettes, et de deux méthodes, manger et penser. Une fourchette sera quant à elle définie à l'aide d'un attribut, son nom, et de deux méthodes, prendre et poser.

M2 est le niveau contenant les méta-méta-données, c'est-à-dire la description de la structure et de la sémantique des méta-données. Ces définitions sont regroupées au sein de méta-modèles. Ces méta-modèles peuvent être perçus comme des langages de description de différents types de données. Dans le cadre de notre exemple, ce niveau contient la définition de ce qu'est une classe, une opération, un attribut ou une association. Toujours de manière simplifiée, une classe est définie comme ayant un nom et deux collections, ses attributs et ses méthodes. Puis un attribut est défini comme ayant un nom et un type, *etc.*

M3 est le niveau décrivant la structure et la sémantique des méta-méta-données. Ces descriptions sont regroupées au sein de méta-méta-modèles. Elles représentent un langage pour définir les différents types de méta-méta-données. Le méta-méta-modèle est souvent figé (codé en dur). Il définit la mécanique de support des contructions de méta-modélisation, par exemple les méta-classes et les méta-attributs.

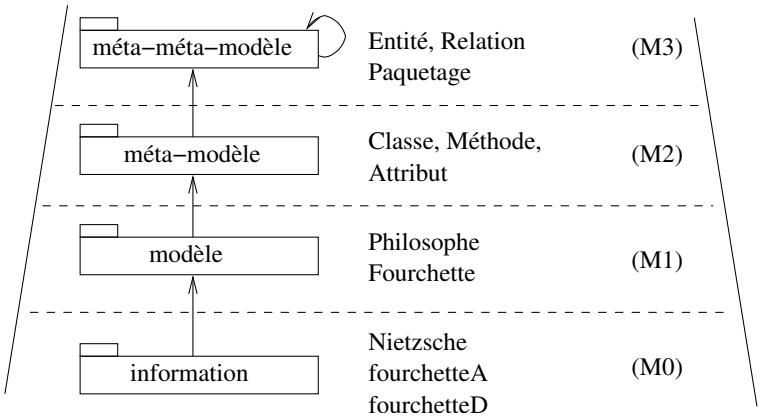


FIG. 4.1 – *Architecture à quatre niveaux du MOF*

Le découpage en quatre niveaux suffit aux utilisations envisagées du MOF. Pour éviter une infinité de niveaux, le niveau M3 est méta-circulaire, il suffit à se décrire. Dans la figure 4.1, les flèches ont pour sémantique « est instance de » ; une méta^n-donnée est instance d'un méta$^{(n+1)}$-modèle. Nous n'avons présenté dans ce schéma qu'un modèle instance d'un méta-modèle, mais l'intérêt est de pouvoir définir une multitude de modèles à partir d'un

méta-modèle. Toujours pour faire le parallèle avec les langages à classes, on ne définit pas un langage par application : le langage Java, par exemple, est utilisé pour produire une multitude d'applications.

La comparaison entre le MOF et les langages de programmation souligne une différence majeure de démarche [16]. Le MOF met l'accent sur les concepts manipulés. La syntaxe est hors propos et ne vient que dans un second temps pour définir des méta-modèles. Il est donc possible d'utiliser plusieurs syntaxes pour mettre en œuvre un méta-modèle. Dans le cas des langages de programmation ou de description (comme les ADLs) la syntaxe est plus visible que les concepts : un utilisateur sera dans un premier temps confronté à une BNF du langage plutôt qu'aux concepts sous-jacents. De plus, les concepts présents au sein d'une BNF ne sont que peu structurés et leur relations ne sont pas nécessairement évidentes.

4.2.2 Le Modèle MOF

L'utilisation du terme « Modèle » avec une majuscule fait référence au méta-méta-modèle du MOF. Le Modèle MOF fournit un certain nombre de concepts de base (plus de vingt), dont nous ne présenterons dans cette section que les cinq qui nous semblent les plus importants et que nous utiliserons par la suite :

- *classe* (dans la section précédente nous avons utilisé le terme d'entité pour simplifier le discours) ;
- *association* (les relations de la section précédente) ;
- *type de données* ;
- *référence* ;
- *package*.

4.2.2.1 Classe

Les *classes* représentent la description des types de méta-objets, les instances de première classe. Les classes sont principalement composées de trois types d'éléments structurels : les *attributs*, les *opérations* et les *références*. Elles peuvent aussi contenir d'autres éléments d'ordre structurels, mais nous ne les discuterons pas ici.

- Un *attribut* définit un contenant pouvant stocker une valeur d'un type de base, ou d'une autre classe. Un attribut peut être déclaré comme modifiable, hérité d'une surclasse et avoir une arité supérieure à un.
- Une *opération* représente un point d'entrée permettant d'accéder au comportement associé à une classe. Elle ne spécifie pas le traitement ni les méthodes qui implanteront ce comportement, mais précise uniquement le nom et les paramètres requis pour invoquer le traitement.
- Une *référence* définit une relation entre la classe et une autre classe. Les références sont discutées un peu plus longuement dans la section 4.2.2.3.

Dans le cadre du MOF, la relation d'héritage entre classes peut être multiple. Cette relation d'héritage, tout comme en UML, est considérée comme une généralisation de(s) la classe(s) héritée(s). Une sous-classe hérite de tous les éléments (attributs, opérations, *etc*) et du comportement de ses super-classes. Au niveau M1, des instances d'une classe A définie

au niveau M2 sont substituables pour des instances de toute super-classe de A. Pour garantir un sens à la définition d'une classe et pour permettre sa projection vers des modèles technologiques, le MOF pose certaines conditions sur l'utilisation de l'héritage.

- Une classe ne peut hériter d'elle-même, que ce soit directement ou indirectement.
- La surcharge des attributs et des opérations n'est pas possible dans une relation d'héritage.
- Dans le cas de l'héritage multiple, une classe ne peut pas généraliser deux classes contenant des éléments portant le même nom. La seule exception est la règle du diamant, où une classe A peut hériter de deux classes B et C contenant un élement portant le même nom, si celui-ci provient d'une même classe D héritée par B et par C.

Enfin, le MOF permet la définition de classes abstraites, ne pouvant être instanciées au niveau M1 et dont l'intérêt est de structurer un méta-modèle à l'aide, entre autres, de relations d'héritage. Cette notion de classe abstraite est donc à rapprocher de la notion du même nom dans les contextes d'UML et de Java.

4.2.2.2 Association

Les *associations* représentent, dans le cadre du MOF, la construction servant à mettre en relation deux classes d'un méta-modèle. Une association définit donc une relation (un lien) entre plusieurs instances du niveau M1. Conceptuellement, ces liens n'ont pas d'identité et ne peuvent donc pas contenir d'attributs ou d'opérations. Toute association MOF contient exactement deux terminaisons (*association ends*) décrivant les extrémités d'un lien. Les types de ces terminaisons sont nécessairement des classes et leur arité peut être multiple. La multiplicité d'une terminaison ne s'applique pas au lien, mais bien à cette terminaison. On peut donc avoir au niveau M1 une association mettant en relation une instance d'un type A et n instances d'un type B. Enfin, une association peut être navigable, au travers de la mise en œuvre de références, et modifiable.

4.2.2.3 Référence

Le Modèle MOF offre deux moyens pour modéliser des relations entre classes. Bien que du point de vue de la modélisation de l'information, les attributs et les relations semblent similaires, les capacités de traitements associées sont différentes. Les associations offrent un modèle d'interactions orienté requête, les traitements sont réalisés sur un objet qui encapsule une collection de liens. Il est donc possible d'appliquer des requêtes sur un type de liens et non sur un lien particulier. Les attributs offrent un modèle d'interactions orienté navigation au travers d'opérations de type *get* et *set* sur un attribut. Cette approche est souvent plus simple et plus naturelle à utiliser que les associations, mais appliquer une requête de recherche globale est très consommatrice en traitements. Il est donc important de choisir entre une association et l'utilisation d'attributs, selon les besoins de manipulation des modèles.

Afin de profiter à la fois des avantages des attributs et des associations, le Modèle MOF fournit un troisième concept : les *références*. Dans la classe où elle est définie, une *référence* se compose d'un nom, d'une terminaison d'association dite « exposée » qui est du même type que la classe dans laquelle elle est définie, et d'une terminaison d'association dite « référencée » qui correspond à l'autre extrémité de l'association considérée. La définition d'une association

dans une classe résulte en un jeu d'opérations, similaires aux *get* et *set* des attributs, qui offriront l'accès et la mise à jour de la projection de l'association au niveau M1.

4.2.2.4 Type de données

Dans la réalisation de méta-modèles, il arrive souvent que l'on ait besoin d'utiliser des types, pour les attributs et les paramètres d'opérations par exemple, qui ne soient pas des types d'objets (*i.e.* ayant une identité). Pour répondre à cela, le MOF définit le concept de *type de données* (*data-type*) qui peut être utilisé pour représenter des types de base comme les chaînes ou les entiers, des types construits comme les structures ou les énumérations, ou encore des types « externes », c'est-à-dire *via* des interfaces qui ne sont pas du type des spécifications MOF.

4.2.2.5 Package

Le *package* représente la construction du Modèle MOF permettant de regrouper des éléments afin de définir des méta-modèles. Au niveau M2, les packages permettent de structurer la définition d'un méta-modèle. Un package peut contenir aussi bien des classes, associations, data-types, *etc* que des packages. Au niveau M1, les packages sont aussi représentés par des instances agissant comme des conteneurs de méta-données. Ces instances représentent les points d'accès aux définitions contenues dans un package, et indirectement en reflète leur visibilité. Le Modèle MOF fournit quatre constructions pour la composition de méta-modèles : la *généralisation*, l'*imbrication*, l'*importation* et le *regroupement*.

Généralisation Tout comme les classes, les packages peuvent être *généralisés* en utilisant une relation d'héritage potentiellement multiple. Lorsqu'un package hérite d'un ou de plusieurs autres, il acquiert tous les éléments de méta-modèles définis dans le(s) package(s) dont il hérite. Au niveau M1, l'instance représentant un package fils peut créer et gérer ses propres collections d'instances de classes et de liens, aussi bien pour ceux définis dans le package que pour ceux hérités. La relation entre instances de super- et de sous-package est la même que la relation entre instances de super- et sous-classe. En cela, l'instance d'un super-package peut être substituée par une instance d'un de ses sous-packages ; mais une instance d'un sous-package est indépendante des instances de ses super-packages.

Imbrication De manière similaire aux *inner class* en Java, un package MOF peut être *imbriqué* (*nested*) dans un autre package, lui-même contenu dans un autre, *etc.* . La sémantique des éléments contenus dans un package imbriqué peut être fortement liée à ce dernier. Il y a quelques restrictions quant à la composition de ce type de package. Un package imbriqué ne peut étendre ou être étendu par un autre package. De même, il ne peut être importé ou regroupé. De tels packages ne peuvent être directement instanciés au niveau M1. Une instance de package imbriqué est dépendante de l'instance du package la contenant ; elle est une composante de cette instance.

Importation Les deux mécanismes précédents permettent l'utilisation globale d'un package par un autre. Le mécanisme d'*importation* fourni par le MOF offre quant à lui un moyen de ne réutiliser que certains éléments d'un package. Un package peut importer un ou

plusieurs autres packages. Dans ce cas, le package qui importe peut utiliser tout élément contenu dans le(s) package(s) importé(s). Il est donc possible, par exemple, de définir une association entre une classe du package qui importe et une classe d'un package importé. Au niveau M1, une instance d'un package n'a pas de relation explicite avec les instances des packages qu'il importe. Elle ne peut donc créer des instances de classes définies dans les packages importés.

Regroupement Le *regroupement* de packages (*clustering*) est une forme d'importation qui lie fortement le package qui importe et le package importé au sein d'un *cluster*. Comme avec la relation d'importation, un package peut *clusteriser* et être *clusterisé* par un ou plusieurs packages. Une instance de package *cluster* se comporte comme si les packages *clusterisés* étaient imbriqués. Les cycles de vie des instances de ces derniers sont fortement liés à l'instance du package *cluster*. Quand une instance du package *cluster* est créée, une instance de chacun des packages *clusterisés* est automatiquement créée elle aussi. Il en va de manière symétrique pour la destruction des instances. Contrairement aux packages imbriqués, des instances de packages *clusterisées* peuvent être créées indépendamment de leur *cluster*.

Expression de méta-modèles Le MOF n'offre pas de syntaxe à proprement parlé pour définir des méta-modèles. Trois moyens sont disponibles pour exprimer des méta-modèles à l'aide des concepts du MOF :

- utiliser la notation graphique UML *Unified Modeling Language* et le concept de profil,
- utiliser le format XMI (*XML Metadata Interchange*) destiné à représenter à la fois des méta-modèles et des modèles sous une forme XML,
- utiliser le langage MODL *Meta Object Definition Language* de définition de méta-modèles [25].

4.2.3 Des modèles aux environnements

Une des caractéristiques intéressantes du MOF est l'ensemble des mécanismes disponibles pour projeter des méta-modèles MOF et produire des environnements supportant leur utilisation, *i.e.* la définition de modèles. Le MOF ne représente donc pas seulement un moyen d'expression, mais aussi un support de l'exploitation de modèles.

4.2.3.1 La projection MOF - OMG IDL

La projection MOF - OMG IDL est un ensemble de patrons *templates* standards permettant de transformer un méta-modèle MOF en un ensemble d'interfaces OMG IDL. Les interfaces résultant de la projection d'un méta-modèle contenant un ensemble de méta-données correspondent aux interfaces des objets CORBA représentant ces méta-données. Les interfaces ainsi produites sont généralement utilisées par un référentiel pour stocker les méta-données [10, 79]. Notre but n'étant pas de présenter les projections de manière exhaustive, nous ne parlerons ici rapidement que des projections concernant les classes, associations et packages.

- Une classe du méta-modèle est projetée en deux interfaces : une d'*objet de méta-donnée* et une de *représentant* (*proxy*) de classe de méta-donnée. Elles supportent les attributs,

opérations et références définies dans le méta-modèle et, dans le cas de la seconde, incluent une opération de type « fabrique d'objets de méta-données ».

- Une association du méta-modèle est projetée vers une interface de *proxy* d'association de méta-données qui supporte les opérations de requêtes et de mise à jour.
- Un package du méta-modèle est projeté vers une interface de *proxy* de package de méta-donnée. Celui-ci agit comme un conteneur pour les *proxies* de classes et d'associations définies au sein du package dans le méta-modèle.

Les règles de projection sont standardisées de manière précise, de telle sorte que des référentiels générés par des outils différents comme M3J [10], dMOF [25] ou RAM3 [79] auront des interfaces identiques pour un méta-modèle donné. En plus des interfaces spécifiques aux méta-modèles, les objets de méta-données partagent des interfaces de base communes pour la réflexion. Ces interfaces permettent à un outil générique de manipuler tout type de méta-donnée sans pour autant être produit par rapport à leur modèle. Cette facilité est à comparer à l'utilisation dynamique d'objets CORBA au travers du DII[2] et du référentiel des interfaces pour produire des outils génériques tels que CorbaScript et CorbaWeb [58, 59].

Le MOF est en pleine évolution avec les travaux sur la spécification 2.0. Un des propositions est de faire évoluer les règles de projection en OMG IDL vers le modèle de composants CORBA : utilisation du langage OMG IDL 3 et de composants pour mettre en œuvre les référentiels.

4.2.3.2 La projection MOF - XML

L'utilité de la projection MOF - XML est la sérialisation de méta-modèles à des fins d'échange entre outils. Le format et les mécanismes d'échange de méta-données MOF sont définis dans une seconde spécification : XML Metadata Interchange (XMI) [74]. La spécification XMI définit deux types de règles de projections. En premier lieu, des règles servent à produire des vocabulaires XML (DTD, *Data Type Definition*, ces règles étant unidirectionnelles (méta-modèle MOF vers DTD XML). Une telle DTD XML fournit un cadre d'échange de méta-données. En second lieu, des règles de production de documents XML définissent une projection bi-directionnelle entre un document XML et des méta-données MOF.

Ainsi, pour un méta-modèle donné, il est possible de générer la DTD XML associée qui servira de base à l'échange de modèles définis à partir de ce méta-modèle. Les modèles seront représentés par des documents XML conformes à cette DTD XML. De plus, effectuer des traitements sur des méta-modèles exprimés graphiquement implique que ces traitements soient inclus dans l'outil graphique. L'utilisation d'un format externalisable permet d'appliquer, sur un méta-modèle, différents types de traitements, par exemple pour faire de la génération d'implantation ou bien une passerelle avec des référentiels n'étant pas conformes avec le MOF. Ceux-ci peuvent être mis en œuvre à l'aide d'outils divers. En addition de cette approche par DTD, le MOF propose aussi une approche utilisant les XML Schemas [102, 103, 104]. Cette dernière a l'avantage d'être plus flexible que l'utilisation de DTD.

2. *Dynamic Invocation Interface*, interface d'invocation dynamique d'objets CORBA.

4.2.4 Conclusion

Le *Meta Object Facility* (MOF) représente la proposition technologique de l'*Object Management Group* (OMG) pour définir et exploiter des méta-modèles. L'aspect technologique de cette proposition souligne le fait que le MOF fournit un ensemble de moyens pour manipuler des méta-modèles, qu'il est une forme de boîte à outils pour la méta-modélisation. Mais, l'OMG ne fournit pas avec le MOF de méthodologie d'utilisation de ces moyens. Il revient à chaque utilisateur de définir sa propre organisation de mise en œuvre du MOF en fonction de ses besoins.

4.3 Séparation des préoccupations

4.3.1 Approche

La séparation des préoccupations (SoC, *Separation of Concerns*) est un concept présent depuis de nombreuses années dans l'ingénierie des logiciels [81, 46]. Les différentes préoccupations des concepteurs apparaissent comme les motivations premières pour organiser et décomposer une application en un ensemble d'éléments compréhensibles et facilement manipulables. La séparation en préoccupations apparaît dans les différentes étapes du cycle de vie du logiciel et sont donc de différents ordres. Il peut s'agir de préoccupations d'ordre fonctionnel (séparations des fonctions de l'application), technique (séparation des propriétés du logiciel système), ou encore liées aux rôles des acteurs du processus logiciel (séparation des actions de manipulation du logiciel).

Par ces séparations, le logiciel n'est plus abordé dans sa globalité, mais par parties. Cette approche réduit la complexité de conception, de réalisation, mais aussi de maintenance d'un logiciel et en améliore le compréhension, la réutilisation et l'évolution. Cette séparation est la motivation d'approches telles que la programmation par aspects [41], les filtres de composition [1], la programmation adaptative [45, 44, 62, 63], la programmation générative [19], ou encore la programmation par sujet [37, 75, 77, 93].

Les exemples suivants illustrent quelques grandes catégories de préoccupations :

- préoccupations concernant les données, avec par exemple la persistance ou le contrôle d'accès ;
- préoccupations concernant les fonctions métiers, avec par exemple les fonctions comptables ou les fonctions de gestion du personnel ;
- préoccupations concernant les règles de gestion, avec par exemple des règles organisationnelles dans une application de gestion de personnel ou des règles de *workflow* dans le suivi du cheminement de l'information ;
- préoccupations architecturales, avec par exemple l'interopérabilité entre applications hétérogènes, des schémas de connexion entre composants répartis ;
- préoccupations système, avec par exemple l'intégration du logiciel métier dans la plateforme d'exécution, la démarcation des transactions, la persistence, la sécurité, *etc.*

La séparation des préoccupations fournit un support méthodologique de modèlisation et de programmation. Elle doit bien sûr être accompagnée d'un processus d'intégration des différents composants générés pour les différentes préoccupations. C'est le processus dit de

tissage (en anglais *weaving*). Ce support permet de gérer plus naturellement et modulairement des intégrations complexes.

4.3.2 Programmation orientée aspects

L'une des techniques les plus étudiées à l'heure actuelle correspond à la *programmation par aspects* (AOP, *Aspect-Oriented Programming*). Celle-ci a été introduite par des chercheurs du XEROX PARC[3] en 1997 [41]. Il s'agit d'une technique novatrice pour l'ingénierie des applications complexes, telles que les applications distribuées.

Elle se fonde sur une séparation claire entre les préoccupations « métiers » (ou « fonctionnelles ») et « non-fonctionnelles » présentes dans les applications. Ce point de vue est similaire à celui présent dans les serveurs d'applications (.Net, EJB, CCM) où les composants fournissent des services métiers et où les serveurs d'applications sont une structure d'accueil proposant aux composants des services système.

Néanmoins, l'AOP ne s'arrête pas à ce premier niveau de découpage, et vise à appliquer la séparation à toutes les préoccupations, qu'elles soient fonctionnelles ou non. Chaque aspect est destiné à être développé de façon indépendante puis intégré à une application par un processus dit de tissage d'aspects (*aspect weaving*). L'une des expérimentations les plus abouties de langage orienté aspect est AspectJ [40] développé par l'équipe à l'origine de l'AOP.

4.3.3 Programmation structurée en contextes

La programmation structurée en contextes, inspirée de la programmation par sujets, vise à supporter la représentation multiple et évolutive d'objets avec points de vue. La motivation principale est d'étudier l'utilisation de référentiels d'objets intervenant dans des contextes applicatifs, ou fonctionnels, multiples. Les motivations sont de proposer une double décomposition, orthogonale, en objets et en fonctions transversales de ces systèmes.

Le projet CROME [99, 100] propose un cadre de programmation par objets structurés en contextes fonctionnels, décrits par des plans. Un plan de base définit la structure du référentiel comme une hiérarchie de classes. Un contexte fonctionnel est décrit par un plan qui adapte les objets du référentiel pour une préoccupation (d'ordre fonctionnelle) particulière.

Chaque plan « fonctionnel » définit un point de vue sur les capacités de traitement du système. Ce point de vue est dédié à une activité particulière. La programmation structurée en contexte, met en œuvre la séparation des préoccupations par la fourniture de points de vue dédiés à des préoccupations particulières. Contrairement à la programmation orientée aspect, il n'y a pas de notion de tissage statique des préoccupations. La séparation des préoccupations existe toujours à l'exécution dans le référentiel.

4.3.4 Séparation multi-dimentionelle des préoccupations

Dans le cas général, la séparation des préoccupations s'articule autour d'une préoccupation centrale. Par exemple, la programmation orientée aspects considère le code métier d'une

3. Le PARC (*Palo Alto Research Center*) nous a entre autre gratifié avant la programmation orientée aspects de Smalltalk, du réseau d'entreprise, de l'interface graphique et de la souris (mise en œuvre des idées novatrices de Engelbart), de l'imprimante Postscript laser et du premier ordiniateur personnel, et ce dès la fin des années soixante.

application comme la préoccupation centrale. Des travaux plus récents menés au centre de recherche T.J. Watson d'IBM proposent une autre vision de cette approche : la séparation multi-dimentionelle des préoccupations. La base de ce travail repose sur les constats suivants.

- De multiples dimensions de séparation peuvent être définies pour un même ensemble de préoccupations.
- Ces multiples dimensions de séparation doivent être utilisables de manière simultannée. Une (ou plusieurs) préoccupation centrale n'est pas imposée aux concepteurs et développeurs. Chaque acteur peut donc disposer de sa propre vision d'un système avec ses préoccupations dominantes. Au-delà de l'identification des préoccupations, il est important que l'encapsulation soit suffisante pour limiter l'impact de l'activité lié à l'ascendance d'une préoccupation sur les autres préoccupations. L'absence d'impact entre préoccupations est considérée comme impossible.
- Percevoir les préoccupations comme indépendantes ou orthogonales est attractif, mais rarement totalement applicable en pratique. Il est donc important de permettre la mise en relation des préoccupations tout en offrant une séparation intéressante.

Le projet *Hyperspaces* [76, 94] expérimente cette approche au travers de la plate-forme *HyperJ*. Un hyper-espace est un espace de préoccupations qui structure la séparation multi-dimentionelle.

- Les préoccupations sont regroupées au sein de dimensions, ce qui donne à *Hyperspace* sa structure multi-dimentionelle. Au sein d'une dimension (un hyper-espace), les préoccupations sont disjointes (pas de définitions communes). Deux préoccupations ne peuvent avoir d'intersection au sein d'une dimension, mais peuvent en avoir si elles sont définies dans deux dimensions distinctes.
- Les dimensions (hyper-espace) sont structurées en modules. Un module contient un certain nombre de concepts représentant une préoccupation ainsi qu'une règle de composition. Ce découpage en concepts et relations provient de la programmation orientée sujets.
- Les modules sont des briques de base et ne représentent pas, en règle générale, des programmes complets ni exécutables. Un système est défini comme un module complet qui peut donc s'exécuter de façon indépendante.

Hyperspaces permet d'identifier explicitement les préoccupations et les dimensions. Il vise à l'inclusion d'artefacts depuis toutes les étapes du cycle de vie d'un logiciel. L'approche par séparation multi-dimentionelle des préoccupations est plus ambitieuse que les approches comme la programmation orientée aspects. Elle est plus générale et ses objectifs sont plus larges. Cependant, il reste encore beaucoup de recherches à faire sur ce sujet pour atteindre ces objectifs.

4.4 *Model Driven Architecture*

La *Model Driven Architecture* (MDA, architecture dirigée par les modèles) [13, 14, 72, 83] est apparue après plusieurs années d'existence de standards de modélisation et de méta-modélisation comme UML (*Unified Modeling Language*) [73] ou le MOF présenté dans la section précédente. La MDA propose une approche qui tend à organiser le(s) modèle(s)

définissant une application ou un système. Pour atteindre cet objectif, la MDA propose un découpage des modèles selon deux préoccupations majeures :

– l'expression des fonctionnalités d'un système ;
– l'expression des spécificités technologiques d'une mise en œuvre de ces fonctionnalités.

Ainsi, associée aux standards technologiques, la MDA permet de mettre en œuvre un même modèle de fonctionnalités à l'aide de solutions technologiques variées. Elle permet aussi l'intégration d'applications en mettant leurs modèles respectifs en relation. Il devient alors possible de supporter l'intégration, l'interopérabilité et l'évolution des applicatifs au fur et à mesure de l'évolution des besoins, et de l'apparition et la disparition des solutions technologiques. En cela, la MDA encourage la *séparation des préoccupations* liées à la modélisation de systèmes informatiques.

4.4.1 Fondements de la MDA

Avant de présenter la mise en œuvre de la séparation des préoccupations dans l'approche proposée par la MDA nous présentons ses concepts de fondamentaux :

– les *modèles* ;
– l'*abstraction*, le *raffinement* et les *points de vue* ;
– la *plate-forme* et l'*environnement* du langage d'implantation.

4.4.1.1 Les modèles

Dans le contexte de la MDA, un *modèle* représente tout ou partie d'une fonctionnalité, de la structure ou encore du comportement d'un système — au sens large, *i.e.* non réduit à du logiciel. La spécification d'un modèle repose sur un moyen ayant une forme bien définie (la syntaxe), une signification (la sémantique) et éventuellement des règles d'analyse, d'inférence ou de preuve. Ainsi, un modèle, au sens de la MDA, doit être associé de façon non ambigue avec un langage de modélisation ayant une syntaxe et une sémantique bien définies, comme celles fournies par le MOF. En cela, un ensemble d'interfaces IDL, un diagramme UML ou encore une version XML de ce diagramme, sont des modèles.

4.4.1.2 Abstraction, point de vue et raffinement

Le terme *abstraction* utilisé dans le cadre de la MDA suit la définition donnée par le modèle de référence d'*Open Distributed Processing* (RM-ODP) [38]. Une abstraction est une réduction (dans un but de simplification) des détails non significatifs. Il est donc important de préciser quels sont les critères d'abstraction qui ont permis la définition du modèle. On parle alors de modèle relatif au *point de vue* défini par ces critères ou, en d'autres termes, une *vue* du système. De plus, la notion d'abstraction peut être abordée à différents niveaux. Ainsi, dans le cas où un modèle est une version simplifiée d'un autre, donc masquant plus de détails, le premier sera considéré à un niveau d'abstraction plus élevé que le second.

A partir de cette dernière assertion, la définition de *raffinement* peut être établie. Un modèle donné –la réalisation– est un *raffinement* d'un autre modèle –l'abstraction– si le premier introduit de nouveaux détails par rapport au second. De plus, la relation de raffinement est elle même décrite comme un modèle, permettant ainsi de garantir que la réalisation est

cohérente et respecte les définitions de l'abstraction. Dans la mise en œuvre, des *points de vue* séparés représenteront, dans certains cas, des niveaux d'abstraction différents. Enfin, alors que la MDA fournit un cadre pour structurer les modèles, les points de vue à utiliser pour un système particulier représente un choix de modélisation.

4.4.1.3 Plate-forme et environnement du langage d'implantation

Dans le contexte de la MDA, la notion de *plate-forme* fait référence à l'ensemble des détails relatifs à la technologie d'implantation utilisée, détails qui ne relèvent pas des fonctionnalités fondamentales ou formelles des composants ou du système. Par exemple, la fonctionalité de créditer un compte bancaire est invariante dans son sens quelque soit le modèle technologique utilisé pour l'implanter, on ajoute dans tous les cas une somme donnée au solde d'un compte.

Ainsi, un modèle indépendant de toute plate-forme technologique représente la spécification de la structure et des fonctions d'un système, indépendamment de toute considération technologique, par abstraction des détails. Une spécification dépendante des artefacts d'exécution d'une plate-forme, l'objet ORB de CORBA, est spéficique à cette plate-forme de mise en œuvre. Chacune des plates-formes considérées dispose elle-même d'une spécification et d'une ou plusieurs implantations. De manière similaire aux plates-formes, les spécifications sont indépendantes des langages d'implantation, comme les souches et squelettes de CORBA en termes de concepts, ou spécifiques à un langage d'implantation, *i.e.* les souches et squelettes générés pour un langage de programmation particulier.

4.4.2 Des modèles, des modèles, des modèles...

La MDA place les modèles au centre de tout processus logiciel, les modèles deviennent des entités de première classe. Afin de faciliter la manipulation de ces modèles, la MDA propose une organisation de ceux-ci.

4.4.2.1 Structuration des modèles de la MDA

La MDA fait une distinction entre certains modèles clés d'un système, introduisant une structuration consistante à ces modèles. Les deux grandes familles de modèles, annoncées dans la section précédente, sont les modèles indépendants des plates-formes (*Platform Independant Models*, PIM) et les modèles dépendants d'une plate-forme (*Platform Specific Models*, PSM). La figure 4.2 présente le méta-modèle de la MDA structurés en terme de PIMs et de PSMs. La réalisation des fonctionnalités d'un PIM est définie par le PSM de manière spécifique à la plate-forme visée. Ceci est dérivé depuis le PIM en suivant certaines règles de transformation.

Les PIMs fournissent les spécifications de la structure et des fonctions du système en faisant abstraction des détails techniques et des considérations technologiques. La MDA définit des relations consistantes entre ces modèles. Ces interactions peuvent être spécifiées à différents niveaux d'abstraction. Au sein de chaque PIM ou PSM, différents niveaux d'abstraction peuvent co-exister. Chaque P*M est une forme de point de vue de la spécification du système. De manière générale, les standards de l'OMG sont spécifiés en termes d'un PIM et d'un ou plusieurs PSMs.

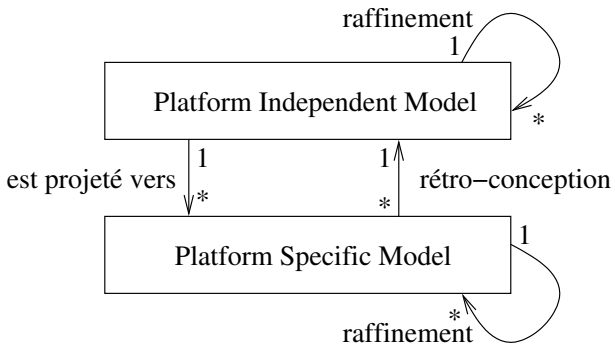


FIG. 4.2 – *Méta-modèle schématique de la MDA*

Abstraire le comportement et la structure, fondamentaux dans un PIM, des préoccupations spécifiques à l'implantation contenue dans un PSM, introduit trois avantages.

- Il est plus simple de valider un modèle non encombré de la sémantique spécifique à une plate-forme. Tous les concepts liés à la plate-forme choisie peuvent être unifiés au niveau du PIM.
- Il est plus simple de produire des implantations pour différentes plates-formes tout en restant conforme aux mêmes structures et comportements fondamentaux du système.
- L'intégration et l'interopérabilité entre systèmes peuvent être définies de manière plus claire en termes indépendants des plates-formes, puis de projection vers des mécanismes propres à ces plates-formes. De cette manière, il est possible d'extraire des projections génériques permettant de transformer automatiquement un PIM vers différents PSMs (que ce soit de manière partielle ou complète).

4.4.2.2 Transformations de modèles

Une fonctionnalité essentielle de la MDA est la notion de transformation. Une transformation regroupe un ensemble de règles et de techniques pour transformer un modèle en un autre. Quatre types sont définis dans la MDA. Elles correspondent aux flèches de la figure 4.2.

PIM vers PIM Ce type de transformation est utilisé pour étendre ou spécialiser un modèle sans ajout d'information dépendant de la plate-forme. Typiquement, cette projection est mise en œuvre pour l'analyse et la conception de modèles. Ce type de transformation est en règle générale lié au raffinement de modèles. Elle est elle même exprimée sous la forme d'un modèle de transformation.

PIM vers PSM Ce type de transformation est utilisé dès lors qu'un PIM est suffisamment raffiné pour être projeté vers la plate-forme d'exécution. Les caractéristiques de la plate-forme servent de base à la projection et doivent être décrites dans un formalisme de modélisation, comme UML par exemple. Le passage d'un modèle abstrait de composants à un modèle technologique comme le CCM représente une projection de type PIM vers PSM.

PSM vers PSM Ce type de transformation est utilisé pour la mise en œuvre de composants et de leur déploiement. Par exemple, la création d'archives de composants se fait par la sélection des services et leur configuration, puis leur installation peut être mise en œuvre en spécifiant la configuration initiale, les machines cibles, la configuration des conteneurs, *etc.* Ce type de projection est en règle générale lié au raffinement des modèles spécifiques à une plate-forme.

PSM vers PIM Ce type de transformation est nécessaire pour abstraire les modèles d'implantation existant dans une technologie particulière vers un modèle indépendant des plates-formes : la retro-conception. Cette transformation est liée à la traçabilité, le « retour » d'un PSM vers un PIM. Le résultat de cette projection ne doit pas aller à l'encontre des règles de la projection PIM vers PSM originale.

4.4.3 Quelques moyens de mise en œuvre

En addition au MOF, l'OMG propose plusieurs solutions technologiques pour définir des modèles. Ces propositions sont une expression de méta-modèles. Le *Unified Modeling Language* et le *Common Warehouse Metamodel* sont des exemples de moyen pour utiliser la méta-modélisation, et d'illustration de sa mise en œuvre.

4.4.3.1 Unified Modeling Language

UML [73] vise la modélisation d'applications, d'objets, des interactions entre objets, des informations relative au cycle de vie des applications, ainsi que, indirectement et à moindre échelle, de certains aspects liés au développement et à l'assemblage de composants. UML peut aussi bien servir à modéliser de nouvelles applications qu'à produire le modèle d'un système existant. Les éléments capturés dans un modèle UML peuvent facilement être exportés vers d'autres outils au cours du cycle de vie d'une application en utilisant le format standard XMI.

Afin de refléter un méta-modèle, la notion de *profil* a été introduite en UML. Un *profil* est une forme de patron introduisant des stéréotypes dans l'utilisation des concepts tels que les classes et les relations. Ces stéréotypes précisent le sens d'une classe ou d'une relation. Différents profils UML sont standardisés comme le profil pour CORBA [66], ou en cours de standardisation comme le profil EDOC [64].

4.4.3.2 Common Warehouse Metamodel

Le CWM [67, 68] représente le standard de l'OMG pour la définition, la manipulation et le stockage de données. Il couvre l'ensemble du cycle de conception, de production et de gestion des applications dans ce domaine, ainsi que la gestion du cycle en soit. L'intégration des outils de développement et de déploiement dans le cadre de conception n'a pas été une préoccupation première de l'OMG pendant de nombreuse années. Désormais, avec l'utilisation des techniques de la MDA, comme les modèles et les DTD XML, de façon transversale à tout le cycle de vie des applications, cette lacune tend à disparaitre.

4.5 Conclusion

Les modèles fournissent des abstractions de systèmes. Du fait de leur indépendance vis-à-vis des technologies de mise en œuvre, la pérennité des systèmes est accrue. La connaissance relative à une application est capitalisée et peut être **utilisée dans le contexte de plusieurs technologies**. De plus, du fait de leur abstraction, les modèles apportent une **simplification dans la compréhension et la manipulation de la connaissance** des systèmes.

Les domaines d'activités étant variés, leurs préoccupations le sont aussi. Il est donc difficile d'envisager un environnement universel de modélisation. Pour répondre à ce constat, **les méta-modèles permettant la définition des concepts d'un domaine d'activité ainsi que de leurs relations**. Ils permettent de disposer d'un **cadre de travail en adéquation avec un domaine d'activité** pour définir des modèles relatifs à ce domaine.

L'*Object Management Group* (OMG) propose le *Meta Object Facility* (MOF) comme boîte à outils de méta-modélisation. Le MOF fournit un ensemble de moyens (concepts et projections) pour définir des méta-modèles et produire les référentiels associés. Ces derniers permettent de définir et manipuler des modèles à l'aide des concepts définis dans les méta-modèles.

La séparation des préoccupations est une approche visant à simplifier la conception et la production d'applications. **Les préoccupations sont les motivations premières pour décomposer et organiser une application en éléments compréhensibles et facilement manipulables**. Chaque acteur dispose des éléments relatifs à sa préoccupation et uniquement de ceux-ci. Ce découpage facilite les activités des différents acteurs du processus logiciel.

La séparation des préoccupations fournit un support méthodologique de modélisation. Elle permet de décomposer la modélisation d'un système en plusieurs modèles dédiés à des préoccupations différentes. Associé à ce découpage, un processus d'intégration des différents modèles est requis. L'association du découpage et du processus d'intégration facilite la collaboration des différents acteurs d'un processus logiciel.

Enfin, la *Model Driven Architecture* (MDA) est un cadre de travail permettant de structurer la spécification de systèmes au travers de modèles. Ces modèles sont ensuite utilisés pour automatiser la production de ces systèmes. L'activité de production de systèmes devient donc avant tout une activité de manipulation de modèles. **Les modèles deviennent des entités de première classe**.

La MDA propose une organisation des modèles en suivant la séparation des préoccupations. La principale séparation est en termes de préoccupations fonctionnelles et de préoccupations techniques. Pour cela, la MDA organise les modèles en deux grandes catégories : les modèles indépendants des plates-formes et les modèles spécifiques aux plates-formes. Au sein de ces deux catégories, les modèles sont structurés par niveaux d'abstraction : du plus général au plus précis. Le transition d'un niveau à l'autre est réalisé par raffinements successifs.

L'utilisation des techniques de méta-modélisation n'est pas encore accessible à tous. La transition entre la conception orientée objets ou composants et l'utilisation de ces techniques représente une étape au même titre que la transition entre l'approche procédurale et les approches objet et composant. De plus, la navigation entre les quatre niveaux de la pile de

méta-modélisation est parfois déroutante et complexe à gérer. Il est nécessaire de prendre du recul pour comprendre les transitions entre niveaux. Toutefois, l'utilisation des techniques de méta-modélisation n'est pas destinée à tous les acteurs d'un processus logiciel. Elle est principalement destinée aux spécialistes d'un domaine d'activité pour fournir un support aux activités de ces acteurs.

Défis Nous avons pu voir dans ce chapitre que les techniques de méta-modélisation répondent bien aux problèmes de conception des systèmes, de représentation des connaissances ou de spécification des activités de *workflow*. Nous nous proposons donc de répondre à la question suivante : **L'utilisation des techniques de méta-modélisation est-elle adaptée et bénéfique dans le contexte des architectures logicielles à base de composants ?** Plus précisément :

- *Comment utiliser les techniques de méta-modélisation pour spécifier les concepts des langages de description d'architectures et leurs relations ?* Donc, de disposer d'ADLs en adéquation avec les processus logiciels.
- *Comment structurer la définition et l'utilisation de ces méta-modèles en suivant une approche par séparation des préoccupations architecturales ?* Donc, de disposer d'ADLs simplifiant l'activité de chaque acteur.
- *De quelle manière de tels méta-modèles pourraient ils être utilisés pour fournir aux acteurs des processus logiciels une méthodologie et des moyens de manipulation des architectures ?* Donc, de disposer d'un cadre de travail facilitant la collaboration des acteurs de processus logiciels.

Deuxième partie

Séparation des préoccupations et méta-modélisation pour architectures logicielles

Chapitre 5

La proposition CODeX

Ce chapitre présente notre proposition CODeX pour définir des ADLs minimaux, extensibles et respectant la séparation des préoccupations. Il présente l'approche CODeX et discute de l'utilisation des techniques de méta-modélisation pour définir un langage de description d'architectures adapté pour un processus logiciel et de sa mise en œuvre au travers d'un environnement de manipulation d'architectures.

- La section 5.1 présente les prémisses et les objectifs de notre travail.
- La section 5.2 présente une vue d'ensemble de notre approche en termes de structuration. La méthodologie de définition d'un langage de description d'architectures et la production de son environnement associé sont discutées.
- La section 5.3 présente notre utilisation des techniques de méta-modélisation de l'*Object Management Group* pour définir le méta-modèle d'un ADL. Elle précise le méta-méta-modèle de CODeX.
- La section 5.4 discute des environnements de manipulation d'architectures associés aux méta-modèles d'ADLs. Elle donne aussi une vision d'ensemble de notre cadre de travail pour produire à partir d'un méta-modèle d'ADL l'environnement de manipulation d'architectures associé.

5.1 Objectifs

Nous avons présenté dans la première partie de ce document un panorama de modèles de composants (cf chapitre 2) et de langages de description d'architectures (ADLs) (cf chapitre 3). De ce panorama, nous avons mis en avant un certain nombre de concepts clés et de limitations de ces deux approches (cf section 2.5 pour les composants et section 3.5 pour les ADLs). Les limitations de ces deux approches auxquelles nous essayons de répondre sont au nombre de quatre.

- Bien que très proches, ces deux approches ne sont pas composables, et il n'est pas possible de choisir le modèle de composants à mettre en œuvre avec un langage de description d'architecture donné.
- Les langages de description d'architectures adressent un nombre limité de préoccupations et ne sont généralement pas extensibles.
- Le méta-modèle des langages de description d'architectures est figé et non explicité, il est uniquement reflété par la syntaxe. De plus, les préoccupations adressées sont mélangées

à un unique niveau (la syntaxe). Ceci réduit l'aptitude des acteurs d'un processus logiciel à collaborer.
- Les langages de description d'architectures ne sont exploités que pendant la phase de production des applications et ne sont pas disponibles pendant la phase d'exécution de celles-ci.

Enfin, le chapitre 4 a présenté les techniques de méta-modélisation proposées par l'*Object Management Group*. Nous avons vu dans ce chapitre que l'utilisation de méta-modèles permet de préciser les concepts d'un domaine d'activité pour disposer de moyens adaptés à la définition des modèles des systèmes de ce domaine. Nous avons aussi vu dans ce chapitre comment la *Model Driven Architecture* met en œuvre la séparation des préoccupations pour structurer la définition et l'utilisation de modèles. L'utilisation de la séparation des préoccupations représente une bonne approche pour faciliter la collaboration des acteurs d'un processus logiciel. Notre travail vise à utiliser conjointement les techniques de méta-modélisation et la séparation des préoccupations pour améliorer la collaboration des acteurs d'un processus logiciel autour des architectures.

Notre proposition **CODeX est à la fois une méthodologie et un cadre de travail pour la définition, la production et l'utilisation d'environnements de manipulation d'architectures** visant à faciliter la collaboration des acteurs d'un processus logiciel. Les quatre objectifs principaux de CODeX sont :
- concevoir des moyens de manipulation d'architectures logicielles qui soient en adéquation avec les besoins des différents acteurs, et fournir les environnements de manipulation associés ;
- mettre en œuvre la séparation des préoccupations afin de structurer la définition et l'utilisation de ces moyens ;
- proposer des solutions qui soient à la fois minimales et extensibles en termes de préoccupations architecturales ;
- enfin, fournir des représentations d'architectures utilisables en phase d'exploitation des applications.

Afin d'atteindre ces objectifs, **CODeX met en œuvre les techniques de méta-modélisation pour définir les moyens de manipulation des architectures**. En parallèle, **CODeX respecte la séparation des préoccupations pour structurer la définition et l'utilisation de ces moyens**. L'approche CODeX repose sur les éléments suivants.
- La conception d'un ADL se fait au travers de la définition de son méta-modèle à l'aide du *Meta Object Facility* (voir section 4.2) de l'*Object Management Group*. Ce méta-modèle définit les concepts des ADLs en fonction du domaine d'activité visé.
- Les méta-modèles des ADLs sont organisés selon un certain nombre de plans distincts, représentant chaque préoccupation architecturale des processus logiciels. Les acteurs de ces processus disposent de plans personnalisés pour la définition et la manipulation des architectures.
- Les méta-modèles d'ADL sont utilisés pour produire de manière automatisée les référentiels destinés à contenir les versions réifiées des architectures. Ces référentiels représentent le cœur des environnements de manipulation des architectures destinés à faciliter la collaboration des acteurs de processus logiciels.

– Ces versions réifiées des architectures sont utilisables tout au long du cycle de vie, de la conception à l'exécution des applications.

5.2 Vue d'ensemble de la proposition CODeX

5.2.1 Description d'architectures et séparation des préoccupations

Un des reproches que nous avons fait aux ADLs dans le chapitre 3, section 3.5, est de ne fournir de réponse qu'à certaines préoccupations, cet ensemble étant en général non extensible. De plus, les préoccupations prises en compte sont toutes mélangées à un même niveau syntaxique. Le méta-modèle d'un tel ADL est rarement explicité et non extensible. La syntaxe mélange l'ensemble des concepts proposés par l'ADL.

La structuration de ces ADLs est donc limitée, ce qui accroît la difficulté du travail de chaque acteur tout en réduisant la capacité des acteurs à collaborer tout au long du cycle de vie des applications. Nous tendons avec CODeX à proposer une réponse à ce problème d'une part, en méta-modélisant les ADLs en fonction des besoins comme discuté dans la section 5.3.1 et, d'autre part, en structurant cette méta-modélisation par séparation des préoccupations comme nous le discutons ici.

La figure 5.1 présente une comparaison entre les ADLs et la structuration proposée par CODeX. Dans le cadre d'un ADL, la partie gauche de la figure, l'ensemble des informations relatives à la description de l'architecture est mélangée à un niveau syntaxique : la structure et la dynamique lorsqu'elle est exprimable, les informations relatives au site d'exécution et à l'implémentation des différents constituants de l'application, les propriétés non fonctionnelles des applications, *etc.*

Dans le cadre de CODeX, partie droite de la figure, nous avons décomposé l'expression de ces différentes informations en trois niveaux contenant un certain nombre de plans [1] pour structurer leur définition et leur utilisation : le *niveau architectural de base* (BAL), le *niveau d'annotation* (AL) et le *niveau d'intégration* (IL). Un plan reflète ici une préoccupation, et sert à regrouper la définition des concepts de cette préoccupation. CODeX utilise aussi un niveau supplémentaire (ML) pour utiliser une architecture dans le contexte d'un modèle technologique particulier.

5.2.1.1 Niveau architectural de base

Le *niveau architectural de base* (BAL, *Base Architectural Level*) de la figure 5.1 est le premier niveau de structuration des préoccupations de CODeX. Ce niveau regroupe les concepts relatifs au cœur des applications et partagés par tous les acteurs de notre processus d'ingénierie du logiciel. **Il définit le vocabulaire commun à tous les acteurs, indépendamment de toute préoccupation**. Dans le cadre actuel de CODeX, où nous visons plus particulièrement les applications réparties à base de composants, c'est à ce niveau que se trouve défini le méta-modèle de composant du domaine. Ce méta-modèle est abstrait dans le sens où il est défini

1. Ce découpage est inspiré des travaux liés à la programmation par objets structurés en contextes (voir section 4.3.3).

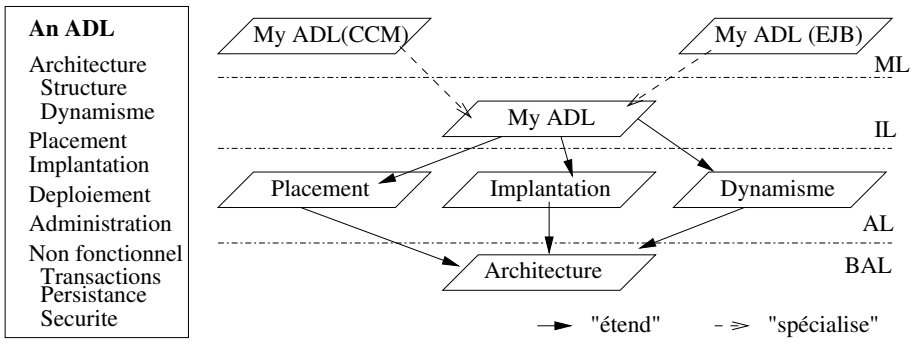


FIG. 5.1 – *Comparaison entre ADLs et niveaux d'abstraction de CODeX*

indépendamment de toute technologie. Cette approche permet de définir le modèle de composant le plus adapté sans être limité par les solutions technologiques actuelles.

Comme nous l'avons vu dans le chapitre 2, la définition du concept de composant logiciel varie d'un contexte à un autre. Il ne nous a donc pas semblé souhaitable d'imposer un modèle de composant particulier, mais plutôt de fournir un moyen pour que les concepteurs d'ADLs puissent définir le modèle de composant le plus approprié. Il sera à la charge du fournisseur de l'ADL de rendre disponible par la suite soit une implémentation propriétaire de ce méta-modèle soit de proposer un moyen pour projeter ce méta-modèle dans un modèle de composant technologique particulier (voir section 5.2.1.4). Ce plan est défini par le concepteur d'ADLs en fonction des besoins applicatifs.

Un second intérêt à cette approche est que la connaissance relative à l'architecture, elle aussi abstraite, d'une application est indépendante de toute technologie. Il est donc ainsi possible d'utiliser cette définition avec plusieurs technologies. Cette approche offre deux avantages. Premièrement, il est possible de ne pas fortement lier la définition de l'architecture avec une technologie et donc, éventuellement, de retarder le choix de la technologie d'implémentation. Ensuite, la capitalisation de cette connaissance rend possible la migration future de l'application vers les technologies à apparaître. La migration, qui sera nécessaire dans le futur pour les applications dont la vie est longue, sera donc moins problématique.

Ce niveau est utilisé par l'architecte pour définir la structure des applications. Dans le contexte des composants logiciels, il peut définir les types de composites et de composants, les instances à utiliser et leur interconnexion. L'expression de l'architecture réalisée à l'aide de ce niveau sera par la suite partagée et enrichie par les différents acteurs. Chaque niveau supérieur disposera donc des concepts définis au niveau de base de la même manière qu'une classe des langages à objets dispose des méthodes et attributs définis dans ses sur-classes. Ce niveau peut donc être considéré, pour les architectures, comme la classe de base d'un langage à objets, par exemple la classe *java.lang.Object* dans le cadre de Java.

5.2.1.2 Niveau d'annotation

Le *niveau d'annotation* (AL, *Annotation Level*) de la figure 5.1 est le second niveau de structuration des préoccupations de CODeX. Ce niveau permet l'enrichissement de la

version de base de l'architecture afin de spécifier les informations relatives aux différentes préoccupations. Afin de poursuivre l'effort de structuration, le niveau d'annotation est composé de différents plans. Chacun de ces plans est dédié à une préoccupation particulière. Ainsi, la séparation des préoccupations est respectée et le nombre de plans sera dépendant du nombre de préoccupations relatives à un contexte applicatif. Cette structuration apporte ainsi une réponse à l'utilisation d'ADLs « sur mesure » ne fournissant des réponses qu'aux besoins.

Chaque plan du niveau d'annotation propose un raffinement de l'architecture de base pour une préoccupation donnée. Pour cela, les plans d'annotation étendent l'architecture de base en annotant chaque concept de base par un ensemble d'informations relatives à la préoccupation du plan concerné. Un plan d'annotation est donc composé de deux parties. (1) Les concepts de ce plan d'annotation sont définis dans un plan de base. (2) Un tel plan hérite du plan de base de l'architecture pour disposer des concepts définis dans ce dernier et utilise le plan de base définissant les concepts relatifs à la préoccupation (cf figure 5.2). Ainsi, les concepts de base architecturaux peuvent être mis en relation avec les concepts de la préoccupation, et donc enrichis. Ces plans sont donc définis conjointement par le concepteur de l'ADL et par des spécialistes de chaque domaine pour savoir quels sont les concepts nécessaires dans le contexte d'utilisation considéré.

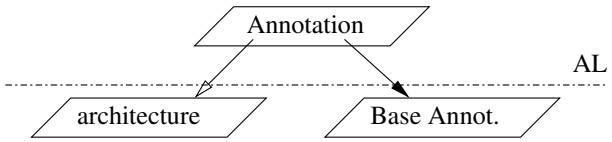


FIG. 5.2 – *Principe d'enrichissement du plan de base architectural*

Le fait de structurer le niveau d'annotation en plusieurs plans permet une mise en œuvre de la séparation des préoccupations. En effet, pour chaque préoccupation un plan est défini de manière indépendante aux autres préoccupations. Par exemple, le fait d'associer une instance de composant avec un site d'exécution ne dépend pas du fait d'associer une instance de composant avec une archive contenant l'implantation de cette instance. D'autre part, il est ainsi possible de définir autant de plans d'annotation que nécessaire (un par préoccupation) et de n'utiliser que les plans requis. Ici encore, la définition de l'architecture est réalisée indépendamment des solutions technologiques qui seront utilisées pour la mise en œuvre de l'application. Ceci tient au fait que l'architecture de base est abstraite et que chaque plan de base est lui aussi défini de manière indépendante des technologies.

Ce niveau est utilisé par chaque acteur autre que l'architecte pour apporter sa contribution à l'édifice. Dans le cadre de notre processus logiciel il existera un plan pour l'expression de la dynamique, un pour l'intégrateur et un pour le placeur. Ensuite, un plan est défini pour chaque préoccupation supplémentaire comme par exemple pour prendre en compte la qualité de service des applications. Dans le cadre de nos expérimentations actuelles, nous n'avons pas mis en œuvre de plan relatif aux propriétés non fonctionnelles comme la persistance, les transactions ou la sécurité. Mais de tels plans peuvent aussi être définis. Une fois tous les plans nécessaires définis, il reste à les intégrer pour fournir une version globale de l'architecture.

5.2.1.3 Niveau d'intégration

Le *niveau d'intégration* (IL, *Integration Level*) de la figure 5.1 est le troisième niveau de structuration des préoccupations de CODeX. **Ce niveau permet l'intégration des différents plans du niveau d'annotation pour fournir une version complète et exploitable de l'ADL**. Contrairement au tissage de l'approche orientée aspects, ce niveau ne fait pas une fusion par écrasement des différentes préoccupations. Il y a conservation de la séparation des préoccupations : il est toujours possible de voir les architectures sous l'angle d'une préoccupation. La non-fusion des préoccupations est une base de notre mise en œuvre de la séparation des préoccupations pour l'ensemble du cycle de vie des applications. Ainsi, cette approche permet de conserver cette séparation lors de l'exécution des applications.

Les moyens disponibles pour la définition d'une architecture à ce niveau sont à mettre en parallèle avec les moyens fournis par un ADL « classique » comme ceux présentés dans le chapitre 3. C'est à ce niveau qu'un ADL défini avec CODeX est disponible et comparable aux ADLs existants. Le principal bénéfice ici est la structuration que CODeX apporte par sa mise en œuvre de la séparation des préoccupations. Tous les concepts requis pour définir une architecture sont disponibles et structurés. De plus, cette version globale de l'architecture est abstraite car toujours exprimée indépendamment des technologies. La définition du plan est quant à elle principalement réalisée par le concepteur d'ADLs.

Pour réaliser l'intégration des plan d'annotation, le plan d'intégration est défini par héritage de tous les plans d'annotation. L'utilisation de l'héritage permet de regrouper tous les concepts provenant à la fois du plan niveau architectural de base et les concepts ajoutés dans les différents plans d'annotation. Comme les concepts provenant du niveau architectural de base ne sont pas modifiés dans les plan d'annotation, ils se retrouvent tels quels dans le plan d'intégration. De même, les annotations introduites au niveau précédent par mise en relation avec les concepts de base se retrouvent au sein de ce plan. Pour chaque concept de base, toutes les relations définies dans chaque plan du niveau précédent sont ainsi associées au concept de base dans le niveau d'intégration.

Ce niveau n'est pas uniquement un niveau d'intégration des plans tels quels. Il est en effet utilisé, par exemple, par le déployeur pour préciser la stratégie de déploiement à mettre en œuvre. Dans notre contexte d'applications à base de composants, il peut être important de préciser l'ordonnancement de déploiement des instances de composants. Encore à ce niveau, l'architecture est abstraite, il en est donc de même pour le processus de déploiement. Ce processus est exprimé indépendamment des particularités de chaque modèle technologique. Le processus de déploiement est exprimé à l'aide d'un ensemble de primitives liées à la dynamique et défini dans le niveau architectural de base. Enfin, ce niveau est disponible à l'exécution pour l'administrateur afin d'avoir accès aux opérations de reconfiguration, elles aussi définies à partir des primitives liées à la dynamique.

A ce point, nous disposons des moyens pour exprimer totalement une version abstraite des architectures des applications visées. Cette version ayant du sens essentiellement pour sa mise en œuvre, il faut maintenant faire le lien avec les modèles de composants technologiques : la projection.

5.2.1.4 Projection vers un modèle technologique

La *projection* d'un ADL abstrait défini en CODeX se fait au travers de la définition de plans au niveau de projection (ML, *Mapping Level*) de la figure 5.1. Comme au niveau d'annotation, il est possible de définir plusieurs plans de projections. Pour chaque technologie de mise en œuvre des applications visée, un plan de projection doit être défini. L'objectif principal de ce plan est de faire le lien entre la représentation abstraite de l'architecture définie par les trois niveaux de CODeX et les instances applicatives, les instances qui offrent le service aux usagers, du modèle technologique choisi.

En addition de ce lien entre représentation de l'architecture et instances applicatives, ces plans ont pour objectif de mettre en œuvre les primitives de la dynamique, définie au niveau du plan de base architectural, dans les modèles technologiques visés. Nous passons donc à ce niveau dans le monde concret des technologies existantes. L'implantation, dans le modèle de composants choisi, des primitives liées à la dynamique va fournir un support à la fois à la mise en œuvre du processus de déploiement, qui pourra de plus être raffiné, et à celle des opérations de reconfiguration pour le modèle. Un exemple de ces mises en œuvre doit répondre à la question : « comment une instance de composant est-elle créée ? »

La définition des plans de ce niveau est réalisée par des spécialistes des modèles technologiques en collaboration avec les concepteurs d'ADLs. Il est en effet requis de comprendre à la fois les concepts définis dans les différents plans de base, architectural et de préoccupation, et de maîtriser les moyens disponibles dans les modèles existants. Pour les concepts définis dans le plan de base architectural et qui n'existent pas dans le modèle technologique, les spécialistes du modèle doivent les mettre en œuvre à l'aide des artefacts disponibles dans ce modèle. Cette partie représente un des aspects les plus techniques de la mise en œuvre de CODeX car il n'est pas possible de fournir une méthodologie pour réaliser cette projection dans tout modèle technologique.

5.2.2 Méthodologie de définition d'un ADL

La méthodologie associée à CODeX répond à la question suivante : « **Comment définir un nouvel ADL ?** » Elle précise comment définir un ADL en fonction du domaine d'activité visé et en respectant le découpage en plans défini dans la section précédente. Cette méthodologie est découpée en cinq étapes.

1. Il faut en premier lieu identifier les préoccupations du domaine d'activité liées aux objectifs du processus logiciel. Cette identification recense tous les concepts importants qui doivent être disponibles dans l'ADL. Deux catégories de préoccupations doivent être identifiées :
   (a) les préoccupations liées aux concepts communs à tous les acteurs : le modèle de composants qui représente le cœur des architectures ;
   (b) les autres préoccupations qui reposent sur ce modèle de composants.
2. Une fois les différentes préoccupations identifiées, il faut les formaliser. Pour cela, il faut tout d'abord caractériser les concepts communs aux différentes préoccupations (le vocabulaire commun à tous les acteurs) et définir le plan de base de l'architecture. Ce plan fournit le modèle abstrait de composants associé à l'ADL.

3. Le plan de base étant défini, il faut pour chaque préoccupation caractériser, d'une part, les concepts de base de cette préoccupation et définir le plan de base de cette préoccupation et, d'autre part, les relations entre ces concepts de base de la préoccupation et les concepts du plan de base architectural.
4. La caractérisation de la composition de ces différents plans donne une version abstraite de l'ADL, c'est-à-dire indépendante de toute technologie de mise en œuvre des applications. Cette caractérisation peut être résumée par l'équation suivante :

   $$\text{ADL} = \Sigma \text{ plans d'annotations}$$

5. Enfin, il faut spécialiser la version abstraite de l'ADL pour sa projection dans une technologie de mise en œuvre des applications. Il est possible de définir plusieurs projections afin d'utiliser un même ADL avec différentes solutions technologiques.

5.3 CODeX et méta-modélisation

5.3.1 Méta-modélisation d'ADLs

La fourniture d'un ADL généraliste et extensible ne nous semble pas la solution la plus adaptée pour répondre aux besoins des acteurs de processus logiciels variés (voir section 3.5). En effet, il peut fournir un noyau de base non adapté aux besoins applicatifs et, d'autre part, des propositions comme xADL (voir section 3.3.2) impose pour chaque nouvelle extension de développer de manière *ad hoc* les outils associés à celle-ci.

Nous avons pu voir dans le chapitre 4 l'intérêt d'utiliser une approche dirigée par les modèles comme la MDA. Nous allons présenter dans la suite de cette section la manière dont nous avons mis en œuvre la MDA pour méta-modéliser des ADL afin d'atteindre notre objectif. Le choix de cette approche représente à nos yeux principalement des avantages :

- disposer d'un ADL en adéquation avec les besoins de chaque domaine applicatif,
- structurer un ADL et son utilisation par la mise en œuvre de la séparation des préoccupations,
- disposer d'un cadre homogène en méta-modélisant ADL et modèle de composants associé,
- bénéficier des règles standardisées d'automatisation de la production d'environnements associés aux méta-modèles comme celles fournies par le MOF,
- enfin, en plus de capitaliser l'information relative aux architectures, l'information relative à la conception des ADLs est elle aussi capitalisée et donc il est possible de les faire évoluer.

Cependant, il est à préciser qu'actuellement, les techniques de méta-modélisation ne sont pas aussi courante d'usage que les techniques de programmation par exemple. Il en résulte donc qu'un inconvénient de notre approche est de mettre en œuvre ces techniques. Toutefois, nous restons confiants vis-à-vis de ce point. Premièrement, seuls les concepteurs d'ADLs ont besoin de les maîtriser, et non l'ensemble des acteurs de notre processus d'ingénierie. Ensuite, ces techniques sont en plein essor comme le montre l'intérêt porté par certains consortiums industriels tel que l'*Object Management Group*.

5.3.2 La pile de méta-modélisation de CODeX

L'utilisation de la méta-modélisation dans le contexte de CODeX est basée sur les techniques de méta-modélisation définies par l'OMG et mises en œuvre dans le cadre du MOF. La structuration verticale du MOF, appelée aussi pile de méta-modélisation, est décomposée en quatre niveaux (voir section 4.2.1.2). La pile de méta-modélisation de CODeX est un peu différente de cette pile MOF à quatre niveaux, puisqu'elle introduit un niveau supplémentaire. Le MOF est destiné à produire des applications. CODeX est destiné à produire des représentations d'applications, leurs architectures.

La pile de méta-modélisation de CODeX est structurée en quatre niveaux comme illustré dans la figure 5.3. Le niveau M3 de CODeX correspond à un ensemble de règles de structuration appliquées à l'utilisation du MOF. C'est le méta-méta-modèle de CODeX. Il propose une spécialisation du MOF pour notre contexte, par restriction aux concepts utilisés. Afin d'introduire de la sémantique, certains packages sont annotés et organisés pour structurer leur utilisation (voir section 5.3.4). Cette utilisation sort du contexte standard du MOF.

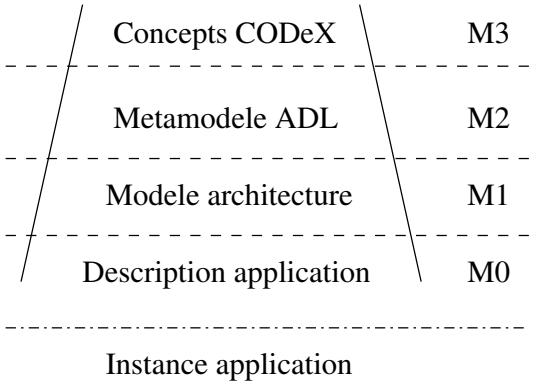


FIG. 5.3 – *Pile de méta-modélisation à quatre niveaux de CODeX*

Les méta-modèles d'ADL sont définis au niveau M2, en respectant les règles définies dans le méta-méta-modèle de CODeX. Ce niveau regroupe la définition de l'ensemble des concepts liés aux environnements de description des architectures : les ADLs. Ne sont définis dans ce niveau que les concepts utiles à un contexte d'utilisation. C'est au travers de la définition de ces concepts qu'il est possible de disposer d'ADLs spécialisés et en adéquation avec les besoins des acteurs du processus d'ingénierie du logiciel. Il peut être défini autant de niveaux M2 que nécessaire. On pourra donc trouver une définition de ce niveau pour les applications de type télécom, une autre pour les applications de type commerce électronique, une troisième pour les systèmes embarqués, *etc.* Ainsi, chacun dispose de son ADL spécialisé : en adéquation avec son contexte d'utilisation et non alourdi par un ensemble de concepts inutiles.

Les architectures des applications, leurs modèles, sont définies au niveau M1. Ces définitions reposent sur les concepts définis au niveau M2. Elles sont similaires à une définition d'architecture telle que le permettent les langages de description d'architectures. Une fois définis, ces modèles d'applications peuvent être utilisés pour créer une ou plusieurs instances des applications. Ils peuvent donc être comparés aux classes dans le contexte des langages de

programmation orientée objet. Dans ce cadre, le mécanisme d'instanciation est défini par le processus de déploiement des applications, c'est-à-dire l'installation sur les sites d'exécution des archives de composants, l'initialisation des serveurs de composants, la création des instances de composants, leur interconnexion et leur configuration.

Le niveau M0 du MOF est peuplé des instances d'applications. Comme nous utilisons CODeX pour définir en premier lieu des architectures, le niveau M0 n'est pas peuplé des instances applicatives, mais des instances représentant les architectures des applications. Celles-ci sont donc à la fois les instances des modèles d'applications, la réification de l'architecture, et les méta-instances des applications, c'est-à-dire la description des instances applicatives qui fournissent les services aux usagers. Pour chaque instance (de composant, de connecteur, *etc*) définie au niveau M1 une représentation est créée au niveau M0. Ces instances de niveau M0 vont représenter l'architecture de l'application qui s'exécute réellement dans le système et contenir les méta-informations utiles au déploiement et à l'administration de cette application.

Enfin, à côté de ces quatre niveaux, les instances applicatives existent dans le contexte d'un modèle technologique de composant particulier. Un intérêt double de CODeX est donc de fournir, d'une part, une représentation standard des applications quelque soit la technologie d'implémentation utilisée et, d'autre part, de ne pas remettre en cause les technologies utilisées puisque les instances applicatives sont utilisées dans le cadre strict de leur modèle, sans extension ni modification. La réification des concepts définis au niveau M2, et non disponibles dans les modèles technologiques, se situera donc au niveau M0 et non au sein des applications. Par exemple, un modèle de composants ne disposant que de références et non du concept de connecteur pourra profiter de ce dernier au niveau de son architecture et donc bénéficier de leur capacité en termes de reconfiguration. A l'exécution, les instances de niveau M1 et M0 coexistent et fournissent une version exploitable des architectures des applications.

5.3.3 Support de la structuration des méta-modèles

5.3.3.1 Définition des plans avec le MOF

Nous avons discuté dans la section 5.2 de la structuration globale d'un ADL sous forme de plans. Pour définir un ADL en respectant cette structuration par séparation des préoccupations, des correspondances doivent être définies entre les termes utilisés dans le chapitre précédent et les concepts offerts par le MOF. Les plans de CODeX servent à structurer le méta-modèle d'un ADL en regroupant des concepts dépendant les uns des autres. L'artefact de structuration offert par le MOF est le *package* qui regroupe un ensemble de définitions. Il est donc logique d'utiliser un *package* MOF pour définir chacun des plans d'un ADL. De plus, les *packages* sont utilisables de différentes façons : imbrication, héritage, import ou regroupement (*cluster*). Cela répond aux besoins vis-à-vis des plans qui peuvent être liés de deux manières : utilisation ou héritage des concepts contenu dans un plan inférieur.

Le méta-modèle d'un ADL est donc défini comme un ensemble de packages contenant les concepts relatifs à une préoccupation. Le package du plan de base architectural regroupe tous les concepts liés au cœur du métier. Ces concepts sont définis par des classes MOF et mis en relation par des associations MOF[2]. Les plans de base d'annotation sont définis de manière

2. Dans le reste de ce chapitre, l'utilisation des termes package, classe, association fera toujours référence à des packages MOF, des classes MOF et des associations MOF si non précisé.

similaire. Les plans d'annotation sont, quant à eux, définis comme héritant du plan de base architectural et important le plan de base d'annotation associé. Ainsi, les concepts définis dans le plan de base architectural sont disponibles tels quels dans le plan d'annotation, de la même manière que s'ils étaient défini directement dans le plan d'annotation ; et les concepts du plan de base d'annotation associé sont accessibles, avec la précision qu'ils proviennent de ce dernier plan. La mise en œuvre de la méta-modélisation correspond donc bien à notre vision de la structuration présentée dans la section 5.2.

5.3.3.2 De l'utilisation de l'héritage de packages

L'héritage de packages n'est pas une notion courante dans le contexte des technologies à base d'objets et de composants. L'usage courant est plutôt de faire l'importation d'un package (en fait de tous les concepts contenus dans ce package) puis de travailler par héritage des concepts dans le but de les étendre de manière unitaire (extension d'un concept à la fois). Cependant, cette approche introduit plusieurs inconvénients. Premièrement, chaque acteur de notre processus manipulerait le même concept avec des noms différents : un composant pour l'architecte deviendrait un composant localisé pour le placeur et un composant implémenté pour l'intégrateur. Pourtant, le cœur du composant défini par l'architecte ne change pas. Ensuite, dans le but d'intégrer toutes les versions enrichies d'un concept, l'héritage multiple serait mis en œuvre. Ceci pourrait poser des problèmes de nomenclature, deux attributs portant le même nom par exemple, qui briseraient la relation d'héritage multiple. En dernier lieu, comment serait nommée la version intégrée des concepts ? Un composant est toujours un composant, mais il serait renommé pour la n-*ième* fois.

L'utilisation de l'héritage de package et de la mise en relation de concepts avec les associations est intéressante pour plusieurs raisons. Premièrement, elle est en harmonie avec notre souhait de permettre le *co-design*, tous les acteurs partagent un concept en utilisant le même nom. Parler de la même chose avec des termes différents est une perte importante de temps et de calme. Toutefois, cela n'empêche pas chaque acteur d'enrichir un concept du plan de base architectural par un concept défini dans un plan de base d'annotation en utilisant une association. De plus, il est possible de définir autant d'associations que nécessaire pour enrichir un concept, donc pas de limitation en termes de nombre de préoccupations prises en compte. Ensuite, l'intégration de toutes les préoccupations est relativement simple. Définir le package d'intégration comme héritant de tous les packages d'annotations implique que tous les concepts présents dans ceux-ci se retrouvent dans le package d'intégration. Ainsi, le concept de composant contenu dans le package d'intégration contient toutes les associations vers les concepts définis dans les différents packages d'annotation. Le concept de base reste inchangé, il est simplement enrichi de toutes les associations relatives aux préoccupations et chaque acteur ne voit au travers de son plan que les associations qui le concernent. Enfin, au delà du fait que la mise en œuvre de cette approche est plus simple que l'importation de packages et l'héritage de concepts, l'utilisation de l'héritage de packages semble mieux réifier notre volonté d'enrichir les concepts de base par séparation des préoccupations.

5.3.3.3 De l'utilisation des associations au niveau d'annotation

Tout comme nous avons défini une utilisation particulière pour l'organisation des packages MOF définissant les différents plans des méta-modèles, certaines règles sont à respecter pour mettre en relation les concepts des différents plans de base au sein des plans d'annotation. Nous avons choisi de mettre en œuvre ces relations entre concepts de base au travers d'associations MOF. Pour respecter la séparation des préoccupations et simplifier sa mise en œuvre, nous avons choisi de ne pas modifier les concepts définis au sein du plan de base architectural. Pour garantir ce point, il est important de suivre certaines règles lors de la définition les associations du niveau d'annotation.

Chaque association est définie avec une référence (cf section 4.2.2.3) afin de faciliter la navigation. Cette utilisation conjointe association / référence modifie la projection des classes MOF en interfaces OMG IDL. Afin de ne pas modifier l'interface du concept provenant du plan de base architectural dans les différents plans d'annotation, une référence ne doit pas être définie sur un concept du plan de base mais sur un concept de plan d'annotation. La seule modification autorisée par rapport au plan de base est la définition d'associations (qui ne modifient pas la projection vers les interfaces OMG IDL). Ainsi, les définitions des concepts du plan de base architectural restent inchangées et l'intégration des plans d'annotation devient triviale. Dans le cadre de nos expérimentations, ce point ne s'est pas illustré comme réduisant les capacité de mise en œuvre vis-à-vis de notre utilisation.

5.3.4 Méta-méta-modèle de CODeX

La figure 5.4 présente le méta-méta-modèle de CODeX. Ce méta-méta-modèle précise la manière dont le MOF doit être utilisé pour structurer la définition de méta-modèles en respectant la séparation des préoccupations. Il est défini au travers d'un certain nombre de « types » de packages.

1. Le « type » de package *ArchitecturalBase* sert à définir l'unique package de base architectural d'un méta-modèle d'ADL.
2. Pour chaque préoccupation deux « types » de packages sont à utiliser.
   (a) *AnnotationBase* sert à définir le package contenant les concepts de base d'une préoccupation.
   (b) *Annotation* sert à définir le package d'intégration des concepts d'une préoccupation au sein de l'architecture. Un tel package doit à la fois hériter du package de base architectural (dans le but de l'enrichir) et utiliser le package de base associé (définissant les concepts de la préoccupation).
3. Le « type » de package *Integration* sert à définir l'unique package d'intégration des différentes préoccupations d'un méta-modèle d'ADL. Ce type de package hérite de tous les packages d'annotation. Il représente l'ADL.
4. Le « type » de package *Mapping* sert à définir une ou plusieurs projections vers un ou plusieurs modèles technologiques de composants. Un package de ce type hérite du package d'intégration d'un ADL.

Cette vision globale du niveau de méta-méta-modélisation dans CODeX reflète la méthodologie (cf section 5.2.2) de définition des méta-modèles d'ADLs en fonction des besoins, tout en respectant la séparation des préoccupations. Ce méta-méta-modèle représente le moyen à utiliser

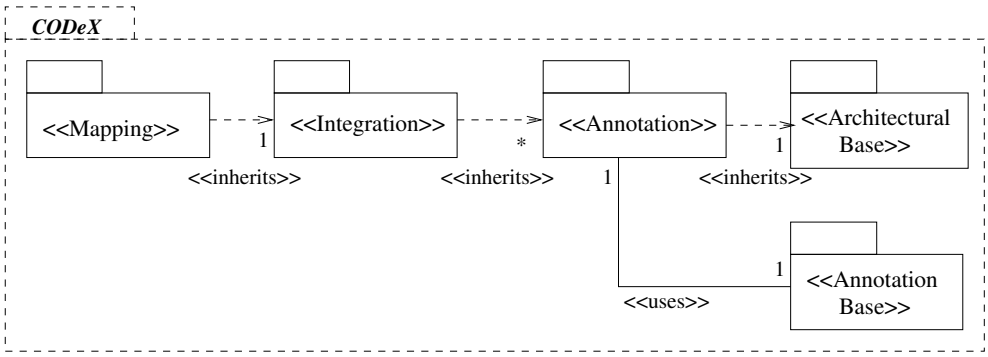


FIG. 5.4 – *Méta-méta-modèle de CODeX*

pour la définition des différents plans d'un ADL. La séparation des préoccupations sert de base à la structuration de la définition des méta-modèles d'ADLs (voir chapitre 6) ainsi qu'à leur utilisation (voir chapitre 7). Une fois le méta-modèle d'un ADL défini, un environnement d'utilisation est à mettre en œuvre. Celui-ci permet à la fois de définir et d'exploiter les architectures. Ces deux activités, lorsqu'elle sont discutées conjointement, seront souvent regroupées sous le terme d'*utilisation* dans la suite de ce document.

5.4 Environnement associé à un ADL

L'approche CODeX tend aussi à fournir des architectures aussi bien lors de la phase de conception que lors de la phase d'exécution des applications. Il est donc nécessaire de disposer d'un environnement supportant cette utilisation. Cette section discute à la fois de l'architecture des environnements, de leur utilisation et de leur mise en œuvre.

5.4.1 Réification des architectures

L'utilisation d'un référentiel de méta-informations représente une base de la réification des définitions d'architectures. Ce référentiel a pour objectif de faciliter la collaboration des acteurs de processus logiciels et de rendre les définitions d'architectures disponibles à l'exécution. Pour supporter la séparation des préoccupations, tout au long du cycle de vie des applications, il est important que ce référentiel respecte aussi cette séparation.

Tout d'abord, l'architecture est centrale dans un processus logiciel (cf chapitre 1). Il est donc important de rendre sa représentation disponible dans un format pivot facilement utilisable. La version réifiée de l'architecture nous semble un format adéquat répondant bien à ce problème. Il est en effet aisé de partager un référentiel entre différents acteurs. C'est par exemple le cas du référentiel des interfaces de la spécification CORBA 2.

Le support de la séparation des préoccupations au sein du référentiel permet de fournir un ensemble de vues sur ce référentiel, chacune d'entre elles étant dédiée aux préoccupations d'un acteur du processus logiciel comme le montre la figure 5.5. Ces vues sont principalement utilisées pendant la phase de définition des architectures. Une fois la structure des applications définie par l'architecte, les différents acteurs vont pouvoir enrichir cette version de base pour

spécifier, par exemple, le placement des instances de composants et les implémentations à utiliser.

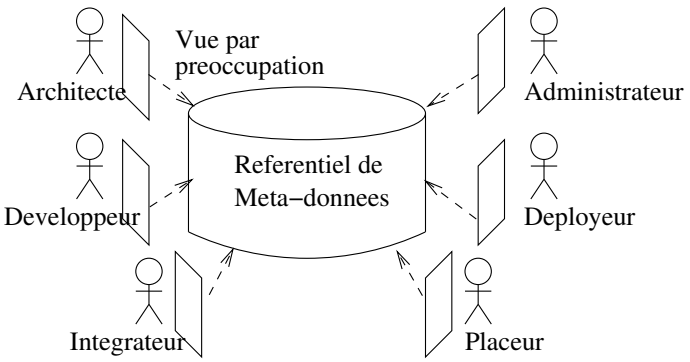


FIG. 5.5 – *Vues par préoccupation du référentiel de méta informations*

Ensuite, lorsque la spécification des architectures atteint une masse critique (en termes d'information requise), les déployeurs peuvent intervenir et préciser le processus de déploiement. Cette activité se situe au niveau du plan d'intégration (voir section 5.2.1.3). Il est en effet nécessaire de disposer de toutes les annotations de l'architecture pour définir la version finale de ce processus. A ce niveau, le référentiel contient non seulement des méta-données –la description de l'architecture– mais aussi des traitements –la mise en œuvre du processus de déploiement. Cette étape représente la transition entre les phases de définition et d'exploitation des architectures. Ceci reflète notre motivation première pour disposer d'une version réifiée des architectures au-delà de la phase de conception : le support du processus de déploiement. Une fois le processus totalement spécifié, le déployeur initie l'évaluation de ce processus et contrôle son déroulement. Une instance de l'application est dorénavant disponible.

Enfin, une fois des instances d'applications existant dans le système, le référentiel conserve tout son intérêt pour les administrateurs. Il contient en effet non seulement des méta-données – la définition des architectures – mais aussi des traitements – la dynamique de ces architectures. L'administrateur dispose donc d'un support à la supervision de l'application ; il a accès à son état, mais aussi à un ensemble d'opérations de reconfiguration définies par le déployeur ou l'administrateur. Au delà de ces opérations de base, il est possible pour l'administrateur de définir des traitements supplémentaires. Ceux-ci peuvent regrouper des opérations prédéfinies utilisées régulièrement de manière groupée ou bien des opérations non anticipées. Il est aussi possible de substituer des instances de composants par des versions plus récentes, par exemple pour la correction d'erreur, ou encore de redéfinir le placement des instances, par exemple pour compenser une panne d'un site d'exécution. Ces deux activités ne mettant en jeu qu'une préoccupation de l'architecture, disposer de la séparation des préoccupations à ce niveau se justifie pour simplifier l'activité de l'administrateur. Ce dernier peut donc intervenir sur l'architecture soit au niveau du plan d'intégration, soit au niveau d'un des plans d'annotation (voir section 5.2.1.2).

5.4.2 De l'utilisation d'un référentiel de méta-données

L'utilisation d'un référentiel de représentation des architectures à l'exécution est discutable. En effet, il pourrait être argumenté que la co-localisation des informations relatives à chaque élément d'une application est une meilleure solution qu'un référentiel centralisé. Tout comme un objet co-existe avec sa classe au sein d'un environnement à objets comme Smalltalk ou Java, la description d'une entité pourrait être réifiée dans le même environnement d'exécution que cette entité. De cette façon, il serait possible d'interroger directement une entité au travers d'un mécanisme réflexif pour découvrir son état et ses capacités de traitement. Sur un autre plan, il serait aussi relativement simple de garantir la cohérence entre l'état d'un élément et sa description de niveau méta puisque toute modification sur le premier serait directement reportable sur la seconde. Enfin, à un niveau plus technique, il serait facilement démontrable que l'exécution d'une méta-opération, par exemple de reconfiguration d'un élément, serait plus performante.

La mise en œuvre de la co-localisation des méta-données avec les entités qu'elles décrivent imposerait de disposer de deux environnements distincts de manipulations des architectures :

- un référentiel pour les phases de conception et de production du processus logiciel ;
- un environnement réflexif pour la phase d'exécution de ce processus.

La prise en compte du processus de déploiement, qui représente la transition entre ces deux phases, peut s'envisager de deux manières :

- un outil extérieur au référentiel contenant l'architecture exploite celle-ci pour réaliser le déploiement ;
- le référentiel inclut les traitements de mise en œuvre du processus de déploiement.

La fourniture d'un référentiel de méta-données regroupant à la fois une réification des architectures et la possibilité de définir des traitements permet de répondre aux besoins de manipulation des architectures, de déploiement et d'administration des applications. Ces traitements représentent la mise en œuvre de la dynamique des applications. **L'architecture est alors réellement au cœur du processus logiciel**.

Cette approche permet de fournir des représentations d'architectures indépendantes de tout modèle technologique, ce qui est plus complexe à réaliser si les méta-données sont intégrées dans l'environnement d'exécution. De plus, pour argumenter sur un plan plus technique, il serait beaucoup plus difficile de fournir, sans référentiel, une vision globale de l'architecture d'une application de manière non intrusive vis-à-vis de l'exécution de cette application. La supervision des applications deviendrait alors une pénalité à leur exécution.

En conséquence, l'utilisation d'un référentiel pour représenter les architectures sous forme réifiée répond bien aux besoins du processus logiciel dans son intégralité. Il permet de fournir une représentation des architectures de manière indépendante à toute technologie (de disposer des PIMs des architectures). La possibilité de définir des traitements dans ce référentiel permet d'une part, de supporter la dynamique des architectures et, d'autre part, d'exprimer des éléments de cette dynamique au-delà de la phase de conception, c'est-à-dire avoir la possibilité lors de l'exécution des applications de définir des actions de reconfiguration non anticipées.

5.4.3 Outillage associé aux référentiels

L'utilisation des référentiels est décomposable en deux phases dont la démarcation est représentée par l'évaluation du déploiement des applications comme nous l'avons discuté dans la section précédente. L'utilisation pré-déploiement des référentiels regroupe toutes les activités liées à la définition des architectures. L'utilisation post-déploiement regroupe quant à elle toutes les activités liées à l'exploitation des architectures. Les deux types d'utilisation des référentiels induisent donc le besoin de deux catégories d'outils.

La première catégorie d'outils relatifs à la définition des architectures regroupe trois types d'usagers. Premièrement, les architectes définissent le cœur des architectures. Ils n'utilisent pas d'informations présentes dans le référentiel si ce n'est celles qu'ils définissent. Ensuite, les acteurs liés à l'activité d'annotation. Ceux-ci utilisent les informations définies par les architectes et introduisent de nouvelles informations. Pour ces deux types d'acteurs une interface graphique interagissant avec le référentiel représente une bonne solution alliant souplesse et facilité d'utilisation. Chaque acteur manipulant un ensemble de concepts prédéfinis mais propres à son activité, un outil graphique générique paramétrable par les concepts à utiliser constitue une solution dédiée et souple. Elle évite d'avoir à produire pour chaque acteur un outil particulier. Enfin, les déployeurs doivent pouvoir préciser le processus de déploiement ; un outil graphique n'est donc pas suffisant. Il est nécessaire de disposer à ce niveau d'un moyen de définir des traitements. Une console dédiée à l'écriture de scripts, par exemple, représente donc un élément important.

La seconde catégorie d'outils relatifs à l'exploitation des architectures regroupe les administrateurs et les déployeurs. Ici encore l'utilisation d'un outil graphique apporte une facilité d'utilisation pour accéder à l'architecture. Une console graphique de supervision est donc une réponse intéressante permettant d'offrir une vision globale de l'architecture d'une instance d'application s'exécutant. Toutefois, cet outil ne doit pas se limiter à l'accès aux informations relatives à l'architecture mais doit également permettre d'invoquer les traitements liés au déploiement et à la reconfiguration des applications. Pour cette dernière activité, nous avons mentionné dans la section précédente que la possibilité de définition de traitements non anticipés était une fonctionnalité intéressante. Ici encore, une console dédiée à la définition de ces traitement est donc nécessaire. Cette console représente en quelque sorte le *shell* des administrateurs Unix.

Les besoins en termes d'outils peuvent donc être résumés comme suit :

- une console graphique d'accès à l'information contenue dans le référentiel, paramétrable par les concepts manipulables, et donc dédiée pour l'utilisation de ces concepts ;
- une console graphique d'ajout d'informations dans le référentiel, paramétrable par les concepts manipulables et donc elle aussi dédiée pour ces concepts ;
- une console graphique d'invocation de traitements disponibles sur l'architecture, paramétrable par les traitements disponibles et dédiée pour ces traitements ;
- une console textuelle de définition de traitements, par exemple sous la forme de scripts.

L'utilisation du terme console et non d'outil est motivée par le fait qu'un outil peut contenir plusieurs de ces consoles. Par exemple, un administrateur a besoin d'une console d'accès à l'information, d'une console d'invocation et d'une console de définition de traitements. Une approche intéressante pour fournir les outils est l'approche composants. Les différentes

consoles seront composées et configurées (paramétrées par les concepts manipulés) en fonction des acteurs et de leurs préoccupations.

5.4.4 Chaîne de production des environnements

Comme nous avons pu le voir dans cette section, l'environnement d'utilisation de CODeX est structuré autour d'un référentiel de méta-informations. Ce référentiel sert à stocker les descriptions d'architectures. Il doit donc permettre de manipuler l'ensemble des concepts définis au niveau du méta-modèle (voir section 5.3.1). Pour obtenir un environnement adapté à une utilisation particulière, celui-ci doit être en adéquation avec le méta-modèle du domaine applicatif visé. Ainsi, la spécification du référentiel par le méta-modèle est une approche répondant aux motivations : le référentiel est réduit aux concepts requis et couvre l'ensemble des besoins. Du point de vue des outils, nous avons plusieurs fois mentionné dans cette section le besoin de paramétrer ceux-ci avec les concepts manipulés par chaque acteur de notre processus d'ingénierie. Ici encore, une partie de l'environnement se retrouve spécifié par le méta-modèle du domaine applicatif. La fourniture de l'environnement repose donc elle aussi sur le méta-modèle considéré. L'automatisation de la fourniture de l'environnement est ainsi un des objectifs de notre approche.

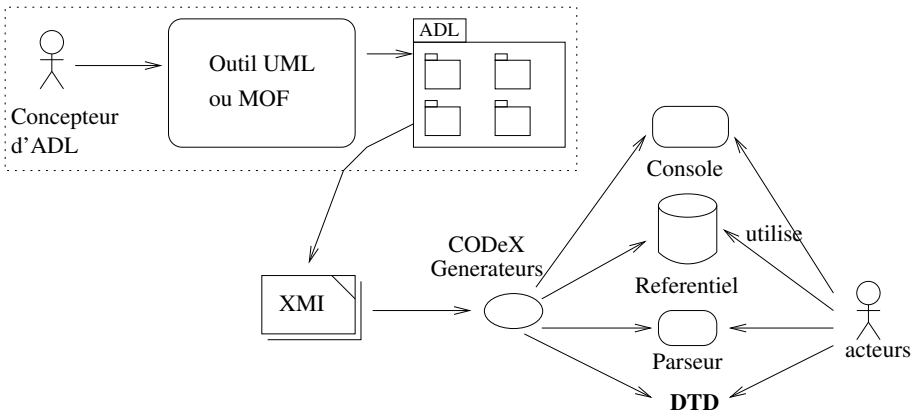


FIG. 5.6 – *De la définition à l'utilisation d'un ADL avec CODeX*

La figure 5.6 présente l'approche suivie pour produire l'environnement d'utilisation d'un ADL à partir de son méta-modèle. Le *Meta Object Facility* (MOF) présenté dans la section 4.2 définit des règles de projection à partir d'un méta-modèle pour obtenir les interfaces d'un référentiel de manipulation de modèles. Cette technique est utilisée dans le cadre de CODeX pour obtenir le référentiel de méta-information à partir des méta-modèles d'ADLs. Une fois le méta-modèle d'un ADL défini et sauvegardé en XMI (*XML Metadata Interchange*), l'outillage CODeX produit un référentiel (interfaces d'utilisation et implémentation) pour définir et exploiter des architectures. Ce point sera discuté plus en détails dans les chapitres 7 et 8.

De manière similaire, la version XMI du méta-modèle peut aussi être exploitée pour définir la DTD des modèles contenus dans le référentiel, toujours selon les règles de projection du MOF. Celle-ci sert alors de base pour disposer d'une version persistante des architectures.

Cette DTD permet de produire des versions XML bien formées des architectures contenues dans le référentiel. Associé à cette DTD, un parseur peut être produit pour (dé)sérialiser les architectures contenues dans le référentiel. Enfin, les consoles d'interactions avec le référentiel sont descriptibles à partir du méta-modèle. Ceci permet la fourniture de consoles spécialisées pour les différentes utilisations requises. Ces éléments n'ont pas été expérimentés dans le cadre de ce travail et ne seront pas discutés dans ce document.

5.5 Conclusion

La motivation de notre travail est de fournir une méthodologie et un cadre de travail pour définir et produire des moyens de manipulation d'architectures qui soient adaptés aux processus logiciels. La mise en œuvre des techniques de méta-modélisation permet de définir le méta-modèle d'un ADL spécialisé pour un processus logiciel. **CODeX fournit un méta-méta-modèle pour la définition d'ADLs**. Celui-ci permet la définition d'ADLs minimaux et extensibles (par extension de leur méta-modèle).

Afin de faciliter la collaboration des acteurs, ce travail repose sur l'utilisation de la séparation des préoccupations pour structurer la définition des architectures logicielles. Elle permet d'offrir à chaque acteur une vision précise de l'architecture par rapport à ses préoccupations. Pour fournir cette structuration, la définition d'un méta-modèle d'ADLs doit suivre un certain nombre de règles. **Le méta-méta-modèle CODeX supporte la structuration par séparation des préoccupations des méta-modèles**.

L'outillage associé à CODeX supporte la production d'environnements de manipulation d'architectures à partir d'un méta-modèle d'ADL. Ces environnements reposent sur l'**utilisation d'une version réifiée des architectures mettant en œuvre la séparation des préoccupations** définie dans le méta-méta-modèle CODeX. Les acteurs du processus logiciel bénéficient donc de cette séparation pour manipuler les architectures au travers de points de vue spécialisés.

L'utilisation d'une **version réifiée des architectures permet de la rendre disponible tout au long du cycle de vie de l'application**, de la conception à l'exécution. Il est ainsi possible d'exploiter l'architecture pour effectuer des tâches comme le déploiement ou l'administration des applications. Cette approche est plus intéressante que les ADLs permettant de définir la dynamique des architectures : il est possible de définir à l'exécution un traitement non anticipé en phase de conception.

Sur un plan plus technique, l'existance de l'héritage de packages dans le MOF peut à première vue soulever des interrogations quant à son usage. Hériter d'un package ou d'un espace de nommage n'est pas une activité possible dans les langages de programmation comme Java ou C++. Cette possibilité peut donc être déroutante. Toutefois, l'utilisation de l'héritage de packages dans CODeX prouve que c'est une facilité intéressante. Elle supporte la définition d'une méthodologie d'utilisation du MOF qui respecte la séparation des préoccupations. L'utilisation de l'héritage de packages contribue à la structuration des méta-modèles.

Les trois derniers chapitres détaillent les trois points suivants.

– Le chapitre 6 discute la définition d'un méta-modèle d'ADL, en mettant en œuvre la méthodologie CODeX et son méta-méta-modèle.

– La production du référentiel associé à l'aide de la mise en œuvre des projections du MOF est présentée dans le chapitre 7.
– Enfin, l'utilisation de ce référentiel dans le contexte spécifique du *CORBA Component Model* est discutée dans le chapitre 8.

Chapitre 6

Méta-modélisation d'un ADL avec CODeX

Le *Meta Object Facility* (MOF) est le moyen de mise en œuvre que nous avons choisi pour définir les méta-modèles des ADLs (cf section 5.3). Nous discutons dans ce chapitre la manière dont le MOF est utilisé pour définir un méta-modèle en le structurant par séparation des préoccupations (cf section 5.3.3). Ce chapitre présente comment les plans sont définis indépendamment les uns des autres à l'aide des artefacts de méta-modélisation fournis par le MOF. L'ADL défini et utilisé pour nos expérimentations est totalement spécifié.

La définition d'un méta-modèle d'ADL est réalisée en suivant la méthodologie associée à CODeX (cf section 5.2.2). Cette méthodologie est ici mise en œuvre pour définir l'ADL utilisé pour nos expérimentations. Ce chapitre présente la définition abstraite d'un ADL, c'est-à-dire indépendante des technologies. Ce méta-modèle permet de définir des PIM (*Platform Independent Models*) d'architectures. Nous ne discutons donc ici que des quatre premières étapes de la méthodologie associée à CODeX. Chaque section discute d'une étape particulière et de l'utilisation du méta-méta-modèle de CODeX.

1. La section 6.1 identifie les différentes préoccupations désirées pour notre ADL.
2. La section 6.2 caractérise les concepts du plan de base architectural et fournit son méta-modèle.
3. La section 6.3 caractérise les concepts de chaque plan d'annotation et définit leur méta-modèle.
4. La section 6.4 caractérise la composition de ces différents plans d'annotation et donne un extrait du méta-modèle résultant.

6.1 Identification des préoccupations

Notre contexte d'expérimentation est la construction et l'exécution d'applications réparties à base de composants logiciels. Une caractéristique de ces composants est qu'ils peuvent être situés sur un ensemble de sites géographiquement répartis. Le déploiement représente la transition entre construction et exécution des applications. Il devient donc une des préoccupations de notre processus logiciel. Enfin, l'ADL que nous définissons dans ce chapitre est destiné à être utilisé selon une approche par prototype comme dans le cadre de Self [98, 89]. Contrairement aux langages à classe, Self n'utilise pas de classes pour définir des objets puis les instancier. Un objet est créé incrémentalement par ajout de méthodes et d'attributs. Pour réutiliser un objet et l'étendre, une opération de clonage est utilisée. Ce choix permet d'illustrer notre propos avec un ADL relativement simple.

6.1.1 Concepts communs à tous les acteurs

Les acteurs identifiés dans l'introduction de ce document (voir section 1.2) manipulent tous des composants et des assemblages de composants. Que ce soit l'architecte pour définir ceux-ci ou les acteurs annotant ces composants pour répondre à une préoccupation. Le terme de composant logiciel est ici utilisé comme nous l'avons caractérisé dans le chapitre 2. Dans le contexte des composants logiciels, un certain nombre de concepts sont nécessairement définis ou à définir pour manipuler de façon précise les architectures. Ces concepts doivent donc être exprimés et spécifiés au niveau du méta-modèle.

En addition au concept de composants, il est important de préciser comment ils sont mis en relation. Le concept de port est important pour exprimer la connectique d'une architecture ; les ports sont aussi bien utilisés pour exprimer les services fournis par un composant que les services utilisés. Toutes les interactions avec un composant sont alors explicites.

Afin de structurer les définitions d'architectures, la notion de composite introduit la notion de hiérarchie de composants. Un composite définit donc un assemblage de composants. Un composite a aussi la propriété d'être manipulable comme un composant. Ainsi, un composite est un assemblage de composants et / ou de composites plus « petits ».

Ces différents concepts représentent le plan de base architectural, c'est-à-dire le vocabulaire commun entre tous les acteurs du processus logiciel.

6.1.2 Mise en œuvre des composants

La préoccupation liée à la mise en œuvre des composants permet de répondre à la question « quoi » déployer pour créer une instance de composant dans un serveur. Une fois l'application déployée, elle fournit de l'information sur les implantations de composants utilisées au sein d'une instance d'application. Cette préoccupation est, elle aussi, importante aussi bien en phase de construction que d'exécution des applications.

Les concepts relatifs à la mise en œuvre des composants sont aussi, et toujours de notre point de vue, au nombre de trois :

- une interface de composant exprime, dans un modèle technologique particulier, le type de composant mis en œuvre ;
- une implantation de composant est une version exécutable d'un composant qui peut être déployée, c'est-à-dire installée sur un site et instanciée ;
- une archive de composant regroupe l'interface du type de composant qu'elle contient ainsi qu'une ou plusieurs implantations de ce type de composant.

6.1.3 Placement des composants

La préoccupation liée au placement des composants permet de répondre à la question « où » un composant doit être déployé. Une fois l'application effectivement déployée, elle permet de répondre à la question « où » une instance de composant est-elle déployée. Cette préoccupation est donc à la fois importante pendant la phase de construction et pendant la phase d'exécution des applications.

Les concepts associés au placement des composants sont au minimum, de notre point de vue, au nombre de trois :

- un site d'exécution représente une machine du système d'information sur lequel les applications doivent être rendues disponibles ;
- une connexion réseau représente un lien entre plusieurs machines du système d'information, leur capacité à communiquer ;
- un serveur de composant représente l'environnement logiciel d'exécution des instances de composants, un tel serveur peut offrir un ou plusieurs conteneurs (cf sections 2.3.2 et 2.4.2).

6.1.4 Dynamique des applications

La préoccupation liée à la dynamique des applications permet de définir comment l'architecture d'une application peut évoluer au cours de son exécution. Ce type d'évolution regroupe un certain nombre d'opérations élémentaires relatives à la création / destruction de composants et à l'établissement / suppression de connexions. La dynamique des architectures est définie au niveau des composites.

Dans le cadre de nos premières expérimentations, nous avons isolé les concepts suivants.

- Une action architecturale regroupe les opération élémentaires à évaluer pour réaliser une modification structurelle de l'architecture. Elle est définie sur un composite et agit sur les instances de composants / composites et connecteurs qu'il contient.
- Les opérations élémentaires de modification de l'architecture sont actuellement au nombre de quatre :
  - l'opération de création d'une instance de composant correspond au déploiement de celle-ci ;
  - l'opération de destruction d'une instance de composant correspond à la suppression de celle-ci, à la fois au sein du composite (sa représentation) et du système (l'instance applicative) ;
  - l'opération d'établissement d'une connexion correspond à la création d'un connecteur entre deux instances de composants et/ou de composites au sein d'un composite ;
  - l'opération de suppression d'une connexion est la destruction d'un connecteur entre deux instances de composants. La suppression d'une connexion n'influe pas sur le cycle de vie des instances mises en jeu.

6.2 Définition du plan de base de l'architecture

Le plan de base de l'architecture contient la définition des concepts de base d'un ADL pour un environnement à composants répartis, c'est le cœur de l'ADL. Cette partie est en fait le méta-modèle de composant utilisé. Ce plan regroupe les concepts utilisés pour définir la structure d'une application.

6.2.1 Caractérisation des concepts

La structure des applications regroupe la définition des services fonctionnels fournis par une application et la manière dont les différents composants logiciels offrant ces services fonctionnels sont assemblés. La définition de la structure doit se faire indépendamment des technologies utilisées afin d'en produire une version abstraite, c'est la partie principale du cœur du PIM de l'architecture.

La réflexion relative à CODeX a été menée dans le cadre des applications réparties à base de composants. Il est donc naturel de trouver le **composant** comme concept central des architectures. Un composant représente ici une unité élémentaire de traitement offrant un certain nombre de services et en utilisant d'autres. L'utilisation de ces services représente les interactions entre instances de composants.

Afin de structurer la définition de ces services et d'expliciter la connectivité des composants, le concept de **port** permet de définir les points de connexion entre composants. Les ports sont de deux ordres : les ports contenant des services fournis et les ports contenant les services requis par une instance de composant. Une instance de composant dispose d'une identité et regroupe ses propriétés configurables, les ports fournis et les ports requis. Un port dispose, lui aussi, d'une identité et regroupe les opérations, fournies ou requises, relatives à ce port. Le cycle de vie des ports est intimement lié au cycle de vie de l'instance de composants les contenant.

Pour construire des assemblages de composants, les ports de deux ou plusieurs instances de composants sont à mettre en relation. Pour cela le concept de **connecteur** est défini. Un connecteur dispose d'une identité et référence les deux ports des instances de composants à mettre en relation. Une connexion est nécessairement établie entre un port fourni et un port requis compatible de deux instances de composants distinctes. Cette compatibilité peut être testé simplement en vérifiant que les signatures des opérations des ports sont compatibles.

Dans le but de structurer les assemblages de composants, le concept de **composite** est défini. Un composite permet de définir un assemblage de composants utilisable comme un composant, en quelque sorte un macro-composant. Ce concept permet de définir des architectures sous forme hiérarchique. Un composite est un assemblage de composants et/ou de composites. Un composite dispose d'une identité et regroupe les instances de composants et les connecteurs assemblant ces derniers. Les cycles de vie des connecteurs et des instances de composants d'un composite sont intimement liés à celui-ci. Tout comme les composants, les composites offrent et utilisent des services fonctionnels au travers de ports. Les services offerts sont réalisés par l'assemblage de composants sous-jacents tel que le présente le patron de conception *facade* [29] : la mise à disposition d'une interface simple malgré la potentielle complexité de mise en œuvre. L'architecture d'une application est donc définie par au moins un composite.

6.2.2 Méta-modèle du plan de base architectural

La figure 6.1 présente le méta-modèle du plan de base architectural de notre expérimentation. Comme discuté dans la section 5.3.3, chaque plan du méta-modèle d'un ADL est défini au sein d'un package MOF. Le plan présenté ici est le plan de base qui est hérité par tous les plans d'annotation, et indirectement par le plan d'intégration. Les différents concepts du niveau du

méta-modèle sont présentés ainsi que leur dépendances. Chaque classe de la figure représente une classe (au sens du MOF). Une flèche représente une relation d'héritage entre deux classes et un trait représente une association (toujours au sens du MOF).

Toutes les classes présentées dans cette figure ont en commun de disposer d'un attribut `name` représentant l'identité de leurs instances. Dans le cas des `ComponentDef` et `ConnectorDef` le nom est un identifiant unique global, vis-à-vis d'une architecture. Dans les cas des `portDef`, `OperationDef` et `ParameterDef` le nom est un identifiant unique relatif, c'est-à-dire unique vis-à-vis de son contenant.

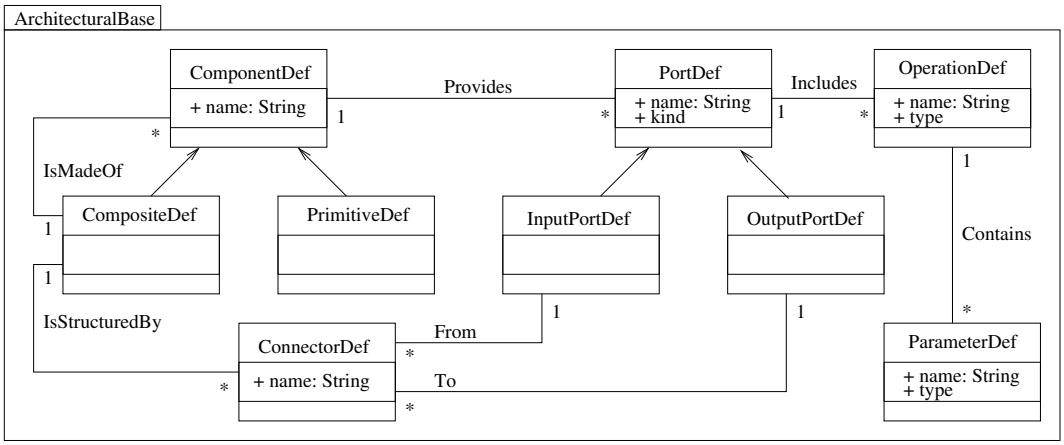


FIG. 6.1 – *Plan de base architectural pour applications réparties*

Le méta-modèle du plan de base architectural regroupe cinq concepts principaux exprimés sous la forme de cinq classes MOF.

- La classe `ComponentDef` représente une instance de composant. Cette classe est spécialisée en `PrimitiveDef`, qui représente un composant primitif et qui sera implanté dans le modèle technologique cible, et en `CompositeDef`, qui représente un assemblage de composants primitifs ou composites. Cette structuration est une mise en œuvre du patron de conception composant / composite.
- La classe `PortDef` est une classe abstraite définissant le concept de port. Un port contient deux attributs : son nom et son mode de communication (synchrone ou asynchrone). Cette classe est spécialisée en `InputPortDef` pour les ports fournis par une instance de composant, et `OutputPortDef` pour les ports requis.
- La classe `OperationDef` représente une opération disponible sur un port (fournie ou utilisée par ce port). Chaque opération contient deux attributs : son nom et son type de retour.
- La classe `ParameterDef` définit le concept d'argument pour les opérations. Elle regroupe deux arguments : le nom de l'argument et son type.
- La connectivité entre les instances de composants contenues dans un composite est exprimée à l'aide de connecteurs. La classe `ConnectorDef` définit ce concept.

Les dépendances entre ces cinq concepts principaux du plan de base architectural sont

définis au travers de sept associations MOF.

- L'association `Provides` met en relation un `ComponentDef` avec un nombre supérieur à un de `PortDef` définissant les ports de cette instance de composant.
- Un port contient un certain nombre, supérieur à un, d'opérations. L'association `Includes` met en relation un port avec les `OperationDef` définissant ses opérations.
- L'association `Contains` met en relation une opération avec ses arguments. Le nombre de ces arguments peut varier de zéro à n.
- L'association `IsMadeOf` met en relation un `CompositeDef` avec toutes les instances de `ComponentDef` (au moins une) qu'il contient.
- L'association `IsStructuredBy` met en relation un `CompositeDef` et les `ConnectionDef` qu'il contient.
- L'association `From` met en relation un `ConnectionDef` avec un port fourni par une instance (`InputPortDef`) et l'association `To` avec un port requis d'une autre instance (`OutputPortDef`). Un port peut être associé à plusieurs connecteurs, mais dans ce méta-modèle, un connecteur définit une association binaire entre deux ports[1].

L'ensemble décrit par un `ComponentDef`, un `Provides`, un ou plusieurs `PortDef`, une ou plusieurs `OperationDef` spécifie entièrement une instance de composant.

Remarques

- Afin de faciliter la navigation entre les différents concepts fortements liés, un certain nombre de références MOF sont définies conjointement aux associations. Par exemple, une référence est définie entre le concept `ComponentDef` et l'extrémité de l'association `Provides` reliée aux `PortDef`. Cette référence permet de demander directement à un `ComponentDef`, et non à l'association `Provides`, la liste de ses ports.
- Les composites respectent, au même titre que les composants, l'encapsulation. Un composite est manipulé comme une boîte noire. Les connexions entre composites se font donc nécessairement via les ports de ceux-ci. Un port de composite représente une mise en œuvre du patron de conception façade [29]. Dans certains cas, ces ports n'auront qu'un rôle de (dé)multiplexeur, *i.e.* permettre la connexion entre plusieurs instances de composants contenues dans des composites différents. Il est aussi possible d'introduire des traitements au niveau de ces ports.
- Dans ce méta-modèle, une instance de composant et un connecteur ne peuvent faire partie que d'un composite à la fois. La définition des associations permet pour un composite de retrouver toutes les instances de composants et connecteurs en faisant partie, et pour chaque composant ou connecteur de retrouver le composite auquel il est rattaché. Il en est de même pour les ports et les instances de composants.

6.3 Définition de plans d'annotation

Notre réflexion s'est focalisée sur trois plans d'annotation : le plan destiné aux intégrateurs, le plan destiné aux placeurs de composants et le plan destiné à la dynamique des applications.

1. La conformité des ports peut être vérifiée dans le référentiel à un niveau syntaxique : les deux ports doivent disposer des mêmes opérations avec les même arguments.

Ces trois plans nous semblent la base de la mise en œuvre du déploiement et de l'administration dans le sens où ils permettent de répondre aux trois question essentielles « quoi », « où » et « comment » déployer. Nous présenterons dans cette section la définition de ces trois plans.

6.3.1 Plans d'annotation d'implémentation

Comme discuté dans la section 5.2.1.2, le niveau d'annotation se décompose pour chaque préoccupation en deux plans. La définition des concepts relatifs à une préoccupation s'effectue au sein du plan d'annotation de base, alors que la mise en relation de ces concepts avec les concepts du plan architectural de base se fait dans le plan d'annotation à proprement parler.

6.3.1.1 Caractérisation des concepts

Dans le but de rationaliser et d'automatiser leur utilisation, les implémentations de composants doivent être rendues disponibles sous une forme manipulable. En règle générale, les implémentations de composants sont diffusées sous la forme d'*archives* dont le format est, pour une technologie donnée, standardisé. Ces archives regroupent les informations relatives aux implémentations de composants. Dans le cadre de notre méta-modèle d'expérimentation nous avons défini une archive comme suit.

- A toute archive est associée une *localisation*, c'est-à-dire l'URL à partir de laquelle il est possible de télécharger l'archive.
- Le contenu d'une telle archive pouvant varier d'un modèle technologique à un autre, nous avons émis comme postulat pour notre contexte qu'une archive contient à la fois :
  - l'*interface* du composant (*i.e.* la définition de son type) ;
  - la (ou les) *implémentation*(s) associées à cette interface. Pour chacune des implémentations, les contraintes relatives sont contenues dans la description de l'implantation : par exemple le fichier du binaire, le langage de programmation utilisé, la version de middleware, le système d'exploitation ou la version de la JVM à utiliser.

6.3.1.2 Méta-modèle des plans d'implémentation

Les plans d'implémentations sont définis au sein de deux packages MOF : le plan de base et le plan d'annotation.

Plan de base d'implémentation La figure 6.2 illustre le plan de base d'implémentation défini au sein du package MOF `ImplementationBase`. Les concepts de bases de la préoccupation implémentation sont au nombre de trois.

- La classe `ArchiveDef` décrit une archive d'implémentation de composant. Elle contient actuellement un unique attribut (`url`) donnant l'adresse où l'archive est disponible pour son téléchargement.
- La classe `ImplDef` décrit une implantation de composant contenu dans l'archive. Cette classe contient différents attributs précisant le langage d'implantation (`lang`), le middleware mis en œuvre (`runtime`), le système d'exploitation ou la JVM cible (`version`), et le fichier contenant la version exécutable de l'implantation (`file`).

– La classe `ItfDef` précise l'interface de l'implémentation (ou des implémentations) de composants contenus dans l'archive. Les attributs de cette classe précisent le nom du fichier contenu dans l'archive (`file`) et le nom de l'interface du type de composant (`name`).

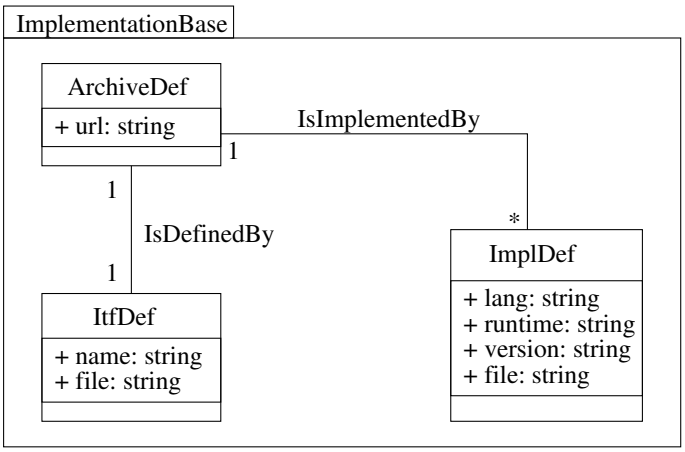


FIG. 6.2 – *Plan de base d'annotation pour les implémentations de composants*

Les dépendances entre ces trois concepts sont définies par deux associations MOF.

– L'association `IsDefinedBy` met en relation l'archive avec l'interface qu'elle contient. Cette association est binaire.
– L'association `IsImplementedBy` met en relation l'archive avec toutes les implantations qu'elle contient. Sa cardinalité illustre le fait qu'une implantation est contenue dans une archive et qu'une archive peut contenir plusieurs implémentations du même type de composant.

Plan d'annotation d'implémentation Le plan d'annotation d'implémentation défini dans le package MOF `Implementation` est illustré dans la figure 6.3. Ce package hérite du package définissant le plan de base architectural (`ArchitecturalBase`) et importe (au sens du MOF) le package `ImplementationBase`. Les classes importées de ce dernier seront nommées avec le nom du package comme préfixe et séparé par le signe '`::`' comme pour la gestion des noms en OMG IDL. Tous les éléments provenant du package `ArchitecturalBase` seront nommés comme dans ce dernier. Le package `Implementation` ne contient qu'une définition : l'association `IsImplementedBy`. Celle-ci met en relation un composant primitif provenant du package de base architectural et une archive définie dans le package de base d'implémentation.

Remarque. Comme dans le plan de base architectural, des références MOF sont définies parallèlement aux associations. Dans le cas des packages d'annotations, une contrainte de définition des références doit être respectée. Afin de ne pas modifier la définition des concepts du plan de base architectural, un référence ne doit pas être ajoutée sur ces concepts. Les références sont définies sur les concepts du plan de base de l'annotation, et « pointent » les

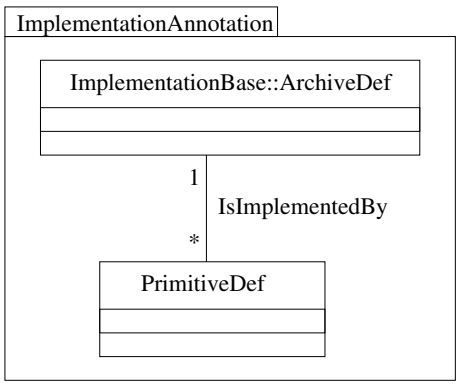


FIG. 6.3 – *Plan d'annotation pour les implémentations de composants*

concepts du plan de base architectural. Ainsi, le vocabulaire commmun des acteurs n'est pas remis en cause.

6.3.2 Plans d'annotation de placement

Après avoir discuté du « quoi » déployer, nous discutons dans cette sous-section du « où » déployer les instances de composants.

6.3.2.1 Caractérisation des concepts du placement

Un modèle de composants est exploitable pour produire des instances d'applications uniquement si ce modèle est associé à un environnement d'exécution (voir chapitre 2). Afin de supporter le déploiement automatique des applications à base de composants, il est important de disposer d'une description de l'environnement d'exécution. Cet environnement est dans notre contexte destiné à supporter des applications réparties. Il est donc composé à la fois d'une partie logicielle, supportant l'exécution des instances de composants, et d'une partie matérielle, les sites d'exécution (les machines), supportant l'exécution de la partie logicielle, et leur interconnexion (le réseau).

Il est nécessaire de disposer, pour chacun des sites d'exécution de l'infrastructure, de leur identifiant (nom ou adresse IP) afin de pouvoir interagir avec elles. Ensuite, il nous semble important d'avoir une connaissance (au moins minimale) sur l'interconnexion des sites d'exécution de l'infrastructure. En d'autres termes, de savoir quels types de réseaux sont disponibles entre quelles machines, et quelles sont les capacités de ces réseaux. Cette connaissance est importante pour optimiser le déploiement des applications en regroupant les instances de composants ayant beaucoup d'interactions sur des sites connectés par un réseau offrant des débits élevés.

Pour chacun des sites d'exécution il est important de connaître les supports d'exécution logiciels disponibles : les serveurs de composants. Ces serveurs offrent un contexte d'exécution aux instances de composants en respectant certaines contraintes. Il faut que le serveur de composants et les instances soient développés dans le même langage, au-dessus du même middleware et, dans certains cas, pour le même système d'exploitation. Donc, comme pour les

implémentations de composants (voir la section 6.3.1), ces contraintes doivent être spécifiées au niveau du serveur.

6.3.2.2 Méta-modèle des plans de placement

Plan de base du placement La figure 6.4 illustre le plan de base du placement défini dans le package MOF `LocationBase`. Ce package définit les trois concepts discutés dans le paragraphe précédent.

- La classe `HostDef` décrit un site d'exécution. Elle contient actuellement trois attributs : l'identifiant de la machine (`id`), le type de cette machine (`type`) et le système d'exploitation supporté ainsi que sa version (`version`).
- La classe `NetworkDef` décrit les supports de communication constituant l'infrastructure. Les deux attributs de cette classe précisent le type de réseau (`kind`) et sa capacité de transit (`bandwidth`).
- La classe `ServerDef` décrit un serveur de composants. Celle-ci regroupe trois attributs fournissant l'identifiant du serveur de composants (`id`), le langage de programmation supporté par ce serveur (`lang`) et la version du compilateur (ou de la JVM) associé (`version`).

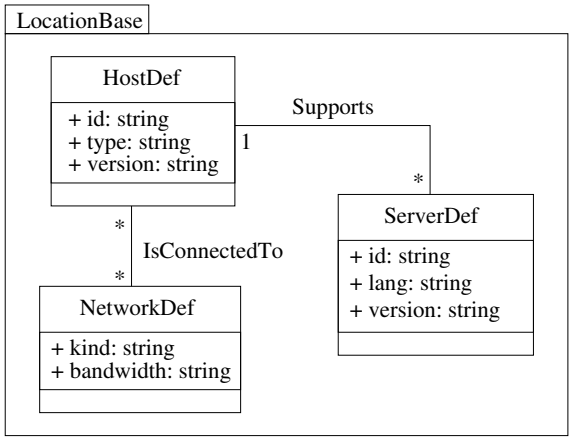


FIG. 6.4 – *Plan de base d'annotation pour le placement des composants*

Les dépendances entre ces trois concepts sont définies par deux associations MOF.

- L'association *IsConnectedTo* définit la relation entre un site d'exécution et les réseaux auxquels il est connecté.
- L'association *Supports* definit la relation entre un site d'exécution et les serveurs de composants qu'il héberge. Un site d'exécution peut héberger plusieurs serveurs de composants, ayant des configuration potentiellement différentes.

Plan d'annotation de placement Le plan d'annotation de placement défini dans le package MOF `Location` est illustré dans la figure 6.5. Comme le package `Implementation`, ce package hérite du package définissant le plan de base architectural (`ArchitecturalBase`) et

importe le package `LocationBase`. Toujours comme le package `Implementation`, le package `Location` ne définit qu'une association, `RunsOn`, entre la classe `PrimitiveDef` provenant du package `ArchitecturalBase` et la classe `LocationBase::ServerDef` importée du package `LocationBase`. La cardinalité de cette association précise qu'une instance de composant ne peut s'exécuter que dans un serveur à la fois et qu'un serveur peut héberger plusieurs instances de composants simultanément.

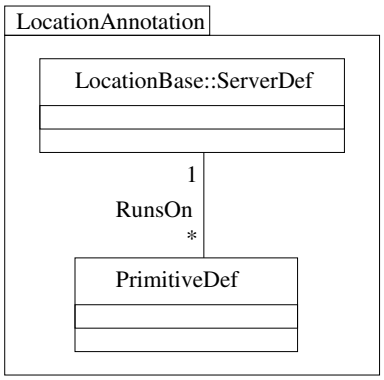


FIG. 6.5 – *Plan d'annotation pour le placement des composants*

6.3.3 Plans d'annotation de la dynamique

La partie dynamique de l'application regroupe les traitements non fonctionnels (vis-à-vis des usagers finaux) de l'architecture permettant d'en modifier la structure, *i.e.* les modifications qui peuvent être apportées sur l'assemblage des composants logiciels. Ici encore, l'expression du potentiel de dynamique d'une application est exprimé indépendamment de toute technologie d'implantation.

6.3.3.1 Caractérisation des concepts de la dynamique

En plus des services fonctionnels décrits précédemment, les composites regroupent un certain nombre de services non fonctionnels (vis-à-vis des usagers finaux). Ces services non fonctionnels sont relatifs à la gestion de la structure du composite, *i.e.* sa dynamique. Ils agissent sur les instances de composants, les connecteurs, ainsi que sur leur assemblage. Les services liés à la dynamique offrent un certain nombre d'actions architecturales comme la création et la destruction d'une instance de composant, l'établissement ou la suppression d'une connexion. Ces opérations représentent la base, à la fois, du support du déploiement d'un composite et de ses capacités de reconfiguration. Tout comme un composant, un composite est instanciable et peut être supprimé du système. Il disposera donc d'une action `deploy ()` et d'une action `remove ()`. Ces actions utiliseront récursivement les opérations relatives à chaque composant, composite et connecteur contenu dans le composite.

L'architecture d'une application est ainsi définie par un graphe d'objets réifiant les éléments de cette architecture. La structure est définie à l'aide des concepts présentés dans la sous-section 6.2 et forme la version initiale de l'application, la version à déployer. Ce déploiement

initial peut être mis en œuvre de deux manières. La première approche est de considérer que tout élément contenu dans un composite est à déployer : il n'est donc pas nécessaire de préciser quelles sont, par exemple, les instances de composants à créer. La seconde approche est d'unifier le déploiement et la reconfiguration et donc de décrire le déploiement initial avec une action architecturale. La première approche apporte la simplicité, la seconde la flexibilité ; les deux sont utilisables dans le cadre de CODeX.

6.3.3.2 Méta-modèles des plans de la dynamique

Nous retrouvons le découpage en deux plans de la préoccupation dynamique : le plan de base et le plan d'annotation.

Plan de base de la dynamique La figure 6.6 présente le méta-modèle du plan de base de la dynamique. Ce méta-modèle regroupe la définition des deux concepts de base de la dynamique.

- La classe `OpBaseDef` permet de définir une opération architecturale élémentaire. Une opération architecturale élémentaire contient la méthode `eval ()` qui entraîne son évaluation. Cette classe est spécialisée en deux catégories d'opérations :
  - `CompOpDef` définit une opération s'appliquant sur une instance de composant ; cette classe est raffinée en deux classes définissant les opérations disponibles sur les instances de composants : les classes `Create` et `Destroy` expriment respectivement les traitements liés à la creation et à la destruction d'une instance ;
  - `CnxOpDef` définit une opération s'appliquant sur un connecteur ; cette classe est raffinée en deux classes définissant les opérations disponibles sur les connecteurs : les classes `Bind` et `Unbind` expriment respectivement les traitements liés à l'établissement et la suppression d'une connexion entre deux composants à l'aide d'un connecteur.
- La classe `ActionDef` permet de définir les actions architecturales relatives à un composite. Une action architecturale est définie comme un ensemble ordonné d'opération architecturales élémentaires. L'opération `eval ()` définie au sein de cette classe permet l'évaluation des traitements architecturaux ainsi définis.

Les deux concepts du plan de base de la dynamique sont organisés au travers de l'association `IsDefinedAs` qui met en relation une action avec les opérations élémentaires qui la composent.

Remarque Les opérations élémentaires agissant sur les instances de composants et sur les connecteurs sont spécialisées pour exprimer des traitements exécutables bien définis. Les quatre opérations présentées nous semblent définir un noyau minimal pour exprimer la dynamique des applications.

Plan d'annotation de la dynamique La figure 6.7 présente le méta-modèle du plan d'annotation de la dynamique des applications. Ce plan définit plusieurs associations, contraitement aux plans d'annotation que nous avons déjà présenté dans ce chapitre. Trois associations relient la dynamique à la structure des applications.

- L'association `Offers` met en relation une `ActionDef` avec le `CompositeDef` pour lequel

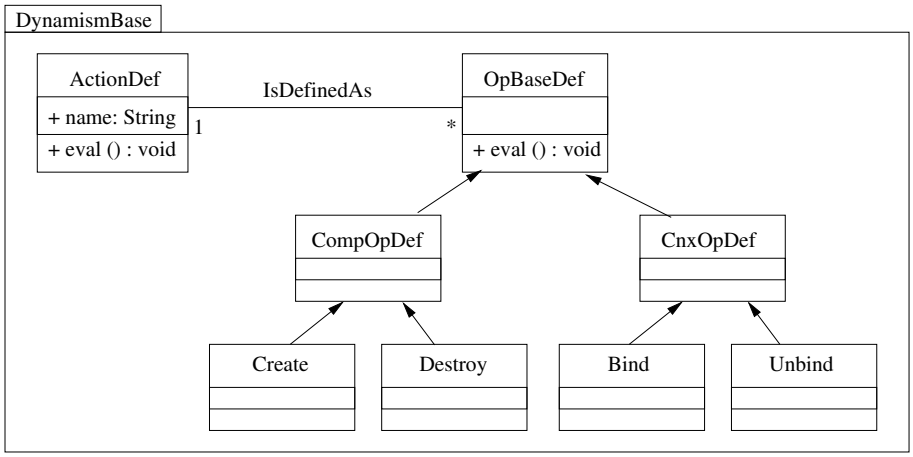


FIG. 6.6 – *Plan de base pour la dynamique des architectures*

elle est définie. Les cardinalités de cette association précisent qu'une action est définie pour un composite donné et qu'un composite peut contenir plusieurs actions.

– L'association `ActsUponCmp` met en relation une opération de type `CompOpBase` avec l'instance de composant sur laquelle elle s'applique.
– L'association `ActsUponCnx` met en relation une opération de type `CnxOpBase` avec le connecteur sur lequel elle s'applique.

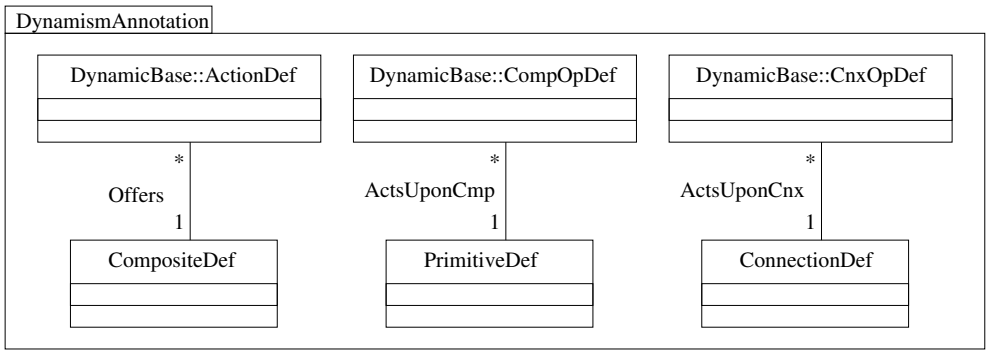


FIG. 6.7 – *Plan d'annotation pour la dynamique des architectures*

Remarques Les associations `ActsUpon*` entre dynamique et structure de l'architecture précisent qu'une opération architecturale élémentaire est en relation avec une seule instance de composant ou un seul connecteur, mais que ces derniers peuvent être contrôlés par plusieurs opérations au sein de la même architecture. Ce choix s'explique par le fait qu'une instance est créée à un moment donné et peut être détruite à un autre. Il est donc nécessaire de pouvoir diriger ces activités par deux opérations distinctes.

6.4 Définition d'un plan d'intégration

6.4.1 Définition des concepts

Au niveau de l'intégration des plans d'annotation, il n'y a pas d'introduction de nouveaux concepts. Ce plan sert à faire une intégration des concepts existants pour produire la version complète de l'ADL. Il est donc normal de trouver à ce niveau une forme comparable aux ADLs existants tels que nous les avons présentés dans le chapitre 3. Contrairement aux plans d'annotation, où il n'y a pas non plus de concepts propres, le plan d'intégration n'est pas destiné à définir de nouvelles associations. Son rôle est uniquement de regrouper les définitions existantes. Ce plan, essentiellement destiné aux déployeurs et aux administrateurs, permet d'utiliser des concepts définis dans les plans inférieurs et de préciser, par exemple, le processus de déploiement abstrait à mettre en œuvre. Ce dernier point est discuté dans la section 6.3.3.2.

6.4.2 Méta-modèle du plan d'intégration

Comme il n'y a ni nouveau concept ni association définis au sein de ce plan, le package associé est « vide » de nouveautés. La figure 6.8 présente le package global d'intégration. Le package d'intégration `MyADL` est simplement défini comme héritant des packages `ImplementationAnnotation`, `LocationAnnotation` et `DynamismAnnotation`, puisque ces trois packages sont les seuls packages d'annotation définis dans le cadre de nos expérimentations. Le package `MyADL` hérite donc de tous les concepts définis dans ces trois packages et par transitivité de tous les concepts définis dans le package `ArchitecturalBase`. `MyADL` contient donc bien tous les concepts de l'ADL visé.

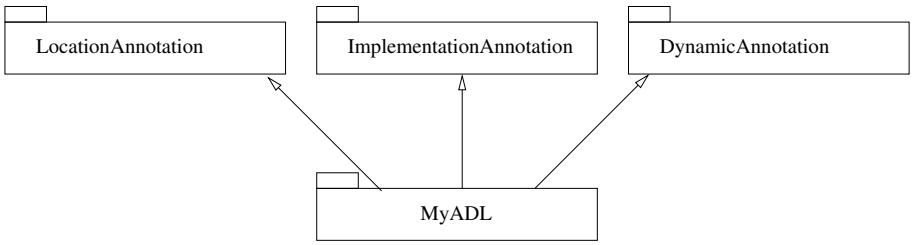


FIG. 6.8 – *Définition du package d'intégration par héritage*

La figure 6.9 présente un extrait du package d'intégration. Cet extrait expose l'intégration de toutes les définitions relatives au concept de composant primitif apparues dans les plans d'annotation : l'association entre le concept de primitif et les concepts de base des préoccupations. Cette figure illustre le fait que tous les enrichissements apportés sur le concept de composant primitif défini dans le plan de base se retrouvent dans le plan d'intégration. Et ceci sans avoir modifié la définition de ce concept. Le concept de composant primitif est ainsi partagé de manière identique par tous les acteurs du processus logiciel.

L'objectif de fournir le méta-modèle d'un ADL en respectant la séparation des préoccupations et facilitant le co-design est donc atteint. De plus, le MOF est un moyen intéressant de définition de tels méta-modèles : les définitions sont exploitables pour produire les environnements associés.

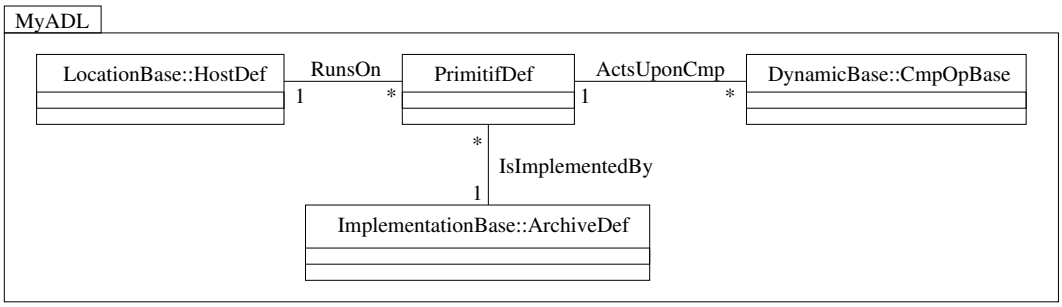


FIG. 6.9 – *Extrait du package d'intégration des préoccupations*

6.5 Conclusion

Nous avons présenté dans ce chapitre la **mise en œuvre de la méthodologie et du méta-méta-modèle de CODeX pour définir le méta-modèle d'un ADL**. Après avoir identifié les préoccupations qui nous intéressaient, l'utilisation de la méthodologie CODeX nous a permis de méta-modéliser les différentes préoccupations et de les intégrer au sein de notre ADL. Ce chapitre montre qu'il est simple d'utiliser CODeX pour définir un ADL sur mesure en fonction des besoins.

Ce chapitre portant sur les quatre premières étapes de la méthodologie CODeX, nous disposons dorénavant d'un moyen pour exprimer des architectures indépendamment des plates-formes d'exécution, les PIMs (*Platform Independent Models*) de la MDA (*Model Driven Architecture*). Il montre donc le fait que l'approche MDA peut être appliquée dans la définition non seulement d'applications, mais aussi de moyens de production des applications.

La MDA encourage à structurer l'activité de modélisation en utilisant des packages pour regrouper les concepts ayant des accointances. La MDA propose de structurer les PIMs en utilisant le raffinement. Les plans d'annotations de CODeX sont des raffinements du plan de base architectural. Nous avons montré ici que les PIMs peuvent aussi être structurés par séparation des préoccupations.

La proposition CODeX représente une utilisation du *Meta Object Facility* (MOF) pour **définir des modèles indépendants des plates-formes en respectant la séparation des préoccupations**. Elle définit une méthodologie d'utilisation du MOF ne reposant que sur ses concepts, donc sans s'écarter du standard. Cette remarque nous semble importante car elle permet d'avancer que l'approche suivie dans le cadre de CODeX n'est pas limitée à cet usage. La modélisation par séparation des préoccupations en exploitant les possibilités du MOF est une approche utilisable dans tout contexte mettant en œuvre la MDA. **La séparation des préoccupations est une approche qui peut aussi être mise en œuvre dans le contexte de la méta-modélisation et qui pourrait être nommé *Aspect Oriented (Meta)Modeling*.**

Disposant du méta-modèle de notre ADL, il est possible de définir des versions abstraites des architectures. Afin de réaliser ces définitions, il et nécessaire de produire l'environnement de manipulation d'architectures associé à notre ADL. Le chapitre 7 présente comment un tel environnement est mis en œuvre. Ensuite, afin d'exploiter les architectures dans le cadre

d'une solution technologique, cet environnement doit être spécialisé pour cette technologie. La spécialisation que nous avons réalisée dans le contexte du *CORBA Component Model* est discutée dans le chapitre 8. Nous présenterons alors comment cet environnement peut être utilisé pour définir, produire et déployer une application.

Chapitre 7

Production de l'environnement associé à un ADL

Ce chapitre présente la mise en œuvre des environnements de manipulation d'architectures. L'accent est mis sur la production des référentiels réifiant les architectures qui représentent le cœur de ces environnements. Nous présentons comment ces référentiels sont produits à partir du méta-modèle définissant un ADL.

- La section 7.1 présente l'outil *CODeX M2 tool* prototypé dans le cadre de cette thèse.
- La section 7.2 présente les projections spécifiées par le MOF pour définir les interfaces des référentiels en OMG IDL à partir d'un méta-modèle. Ces règles de projection représentent la transition entre la définition d'un méta-modèle et son utilisation.
- La section 7.3 discute de notre mise en œuvre de ces interfaces avec le langage OMG IDLscript afin de fournir des référentiels flexibles de manipulation des architectures.
- La section 7.4 présente un bilan de cette mise en œuvre. Elle souligne les avantages de ce choix en termes de souplesse et de concision.

7.1 A propos de l'outillage MOF

Au moment de nos première expérimentations avec le MOF, il n'existait pas d'outils disponibles prenant en charge la production de référentiels mettant en œuvre l'héritage de packages. L'outil dMOF du DSTC (*Distributed System Technology Center*) [25] n'est pas réellement disponible dans sa version XMI. L'outil M3J (*Meta Meta Models in Java*) développé par Xavier Blanc [10] était, quant à lui, limité à l'utilisation d'un seul package MOF par modèle. Nous avons donc développé notre propre outil de projection (vers le langage OMG IDL) et de production des implémentations.

Cet outil ne prend pas en charge tous les concepts du MOF, mais uniquement les concepts dont nous avions besoin dans le cadre de CODeX. C'est un outil *MOF-compliant*, sans être un véritable outil MOF (qui, lui, serait complet). Les interfaces de notre référentiel respectent la spécification MOF pour les concepts qui nous intéressent. Comme pour toute spécification, l'OMG laisse une certaine liberté pour la mise en œuvre des référentiels. Nous avons donc réalisé cette mise en œuvre en fonction de nos besoins. L'outil *CODeX M2 tool* exploite la représentation des méta-modèles en XMI pour générer à la fois les interfaces OMG IDL et l'implémentation en OMG IDLscript [60, 70] des référentiels.

Un second aspect a motivé le choix de réaliser notre propre outillage. Il est important pour certaines préoccupations, comme la prise en charge de la dynamique, de pouvoir disposer de

traitements au sein des référentiels. Les outils comme dMOF ou M3J permettent de répondre à ce besoin, mais de manière statique. Produire des référentiels dans un langage compilé, comme Java, implique que les traitements contenus dans un référentiel sont définis au moment de la production du référentiel ; il n'est alors pas possible d'en ajouter dynamiquement. Le fait d'utiliser un langage interprété, comme OMG IDLscript, permet de définir des traitements *a posteriori* au sein du référentiel. Il est donc possible pour un administrateur de définir, une fois les architectures conçues, des nouveaux traitements, ou de modifier le comportement par défaut de certains traitements contenus dans un référentiel.

7.1.1 *CODeX M2 tool*

L'outil *CODeX M2 tool* a été réalisé sous la forme d'un prototype destiné à expérimenter et valider notre proposition. Cet outil a été implanté en *jIdlscript* [17]. *jIdlscript* est une implantation en Java du standard OMG IDLscript. Le choix d'un langage de script a été motivé par la rapidité et la simplicité d'utilisation pour le prototypage. Le choix de *jIdlscript* plutôt que *CorbaScript* (la version originale en C++) a été motivé par le fait que *jIdlscript* offre accès non seulement à tout objet CORBA mais aussi à tout objet Java. Ceci permet de réutiliser les librairies écrites en Java au sein d'un programme *jIdlscript*, comme la librairie `xerces` offrant une implantation des *APIs DOM* et *SAX* de manipulation de documents XML. La version actuelle de ce prototype représente un peu plus de 4 000 lignes de code pour la production de référentiels indépendamment des technologies.

CODeX M2 tool réalise la production des interfaces OMG IDL du référentiel associé à la définition d'un méta-modèle, ainsi que leurs implantations. Pour cela, notre outil utilise une version XMI des méta-modèles (voir section 4.2.3). Dans un premier temps, le fichier XMI est analysé (cette tâche est réalisée par xerces) pour créer une version DOM (Document Object Model) [101] du fichier XMI. Cette version est ensuite analysée pour créer une représentation interne à notre outil d'un méta-modèle. Cette représentation crée un graphe d'objets typés (contrairement à l'arbre DOM) représentant le méta-modèle et facilitant la navigation.

Deux générateurs sont actuellement intégrés à notre outil : un pour la production des interfaces OMG IDL et un autre pour la production des implantations en OMG IDLscript. Le générateur d'interfaces met en œuvre les mappings définis dans la spécification du MOF. Le générateur d'implantations produit une version adaptée à notre contexte de ces interfaces. Actuellement, seuls les six concepts nécessaires à notre approche sont supportés : package, class, association, data-type, operation et attribut. Ces deux générateurs exploitent la représentation interne d'un méta-modèle construite par le parseur de fichiers XMI.

Le runtime produit par cet outil représente quant à lui un peu plus de 5 000 lignes de code générées dans le cas du méta-modèle défini au chapitre 6 et 500 lignes de mise en œuvre des interfaces relatives à la réflection – package *reflective* du MOF. La mise en œuvre de ce package ne regroupe par tous les traitements relatifs à la réflexivité, mais nous a permis de factorisé un certain nombre d'opérations. La compacité du runtime tient dans le choix d'utiliser un langage de script à typage dynamique et supportant l'héritage dynamique ; plutôt qu'un langage compilé, typé et ne supportant que l'héritage simple comme Java.

7.2 Interfaces OMG IDL des référentiels

Les interfaces du référentiel associé au méta-modèle de l'ADL respectent les projections définies dans le cadre du MOF. Dans ce sens, les référentiels ainsi produits sont compatibles avec le MOF. Nous présentons brièvement dans cette section les trois projections principales utilisées dans le cadre de CODeX : le package, la classe, l'association.

7.2.1 Projection en OMG IDL des packages MOF

7.2.1.1 Projection d'un package de base

Pour tout package MOF défini dans un méta-modèle, la spécification fournit un schéma de projection vers le langage OMG IDL. La figure 7.1 illustre une partie de la projection en OMG IDL du package *ArchitecturalBase*, produite à partir du méta-modèle que nous avons discuté dans le chapitre précédent.

Pour chaque package du méta-modèle, un module est défini afin de regrouper les différentes définitions relatives à ce package. Les schémas de projection des classes et associations seront présentés dans les sections suivantes. Dans le but de pouvoir instancier l'objet représentant le package *ArchitecturalBase*, le patron de conception fabrique [29] est mis en œuvre. L'interface `ArchitecturalBasePackageFactory` offre une unique opération ; elle permet la création de cet objet. Le patron de conception fabrique est utilisé pour l'instanciation de tout objet du référentiel. Pour créer les objets représentant les classes et associations, il faut directement s'adresser à l'objet représentant le package qui les contient.

```
module ArchitecturalBase {
  // définition des classes contenues dans le package
  // définition des associations contenues dans le package

  interface ArchitecturalBasePackageFactory {
    ArchitecturalBasePackage create_architectural_base_package ()
      raises (Reflective::MofError) ;
  } ;

  interface ArchitecturalBasePackage : Reflective::RefPackage {
      readonly attribute ComponentDefClass component_def_ref ;
      readonly attribute CompositeDefClass composite_def_ref ;
      readonly attribute Provides          provides_ref ;
      // autres définitions contenues dans le package
  } ;
} ;
```

FIG. 7.1 – *Projection du package* ArchitecturalBase *en OMG IDL*

L'interface `ArchitecturalBasePackage` hérite de l'interface `Reflective::RefPackage` qui définit son statut de représentante d'un package et fournit des opérations génériques de manipulation des packages. L'interface ainsi définie contient un attribut pour chaque concept défini dans le package *ArchitecturalBase*. Ces attributs fournissent une référence sur la fabrique dédiée au concept. Par exemple, l'attribut `component_def_ref` donne accès à la fabrique

`ComponentDefClass` qui crée des instances de `ComponentDef` (voir § 7.2.2). Cette dernière fabrique sert aussi de gestionnaire d'instances de `ComponentDef` : elle permet de retrouver toutes les instances créées par elle-même, voire dans tout le référentiel (donc par les autres fabriques de ce type), et de détruire ces instances si besoin est.

7.2.1.2 Projection d'un package d'annotation ou d'intégration

Dans le cas de la définition d'un package d'annotation, la projection en OMG IDL est un peu différente. Les relations d'héritage et d'import sont elles aussi exprimées en OMG IDL comme c'est illustré dans la figure 7.2. Dans ce cas, la définition de l'interface `LocationPackage` hérite de l'interface `ArchitecturalBase::ArchitecturalBasePackage`, récupérant ainsi toute les définitions d'attributs contenus dans cette dernière. Les attributs propres au package de placement sont définis explicitement dans cette interface. L'interface de fabrique n'est, quant à elle, pas différente de celle définie dans le package de `ArchitecturalBase`. De plus, le fichier contenant cette définition doit importer avec la primitive `#include` le fichier contenant la définition du module `ArchitecturalBase`.

```
#include <architecturalbase.idl>
module Location {
  // définition de l'association contenue dans le package

  interface LocationPackageFactory {
    LocationPackage create_location_annotation_package ()
      raises (Reflective::MofError) ;
  } ;

  interface LocationPackage
    : ArchitecturalBase::ArchitecturalBasePackage {

    readonly attribute RunsOn runs_on_ref ;
  } ;
} ;
```

FIG. 7.2 – *Projection du package* Location *en OMG IDL*

7.2.2 Projection en OMG IDL des classes MOF

Pour toute classe MOF d'un méta-modèle, la spécification fournit un schéma de projection vers le langage OMG IDL. La figure 7.3 illustre la projection en OMG IDL de la classe *ComponentDef* définie dans le méta-modèle que nous avons discuté dans le chapitre précédent. Les définitions d'interfaces relatives à une classe MOF se retrouvent dans le module associé au package qui contient cette classe dans le méta-modèle. Dans la figure 7.3, les définitions provenant de la projection de la classe MOF *ComponentDef* se retrouvent donc dans le module `ArchitecturalBase`.

La projection MOF vers OMG IDL d'une classe produit deux interfaces : l'interface de fabrique et l'interface définissant la réification du concept. L'interface de fabrique `ComponentDef-`

```
  module ArchitecturalBase {
    // ...
    interface ComponentDef ;
    typedef sequence<ComponentDef> ComponentDefSet ;

    interface ComponentDefClass : Reflective::RefObject {
(a)   readonly attribute ComponentDefSet all_of_type_component_def ;
(b)   readonly attribute ComponentDefSet all_of_class_component_def ;
      ComponentDef create_component_def ()
        raises (Reflective::MofError) ;
    } ;

    interface ComponentDef : ComponentDefClass {
      PortDefSet port_ref () raises (Reflective::MofError) ;
      void set_port_ref (in PortDefSet new_value)
        raises (Reflective::MofError) ;
      void add_port_ref (in PortDef new_element)
        raises (Reflective::MofError) ;
      void modify_port_ref (in PortDef old_element,
                            in PortDef new_element)
        raises (Reflective::NotFound, Reflective::MofError) ;
      void remove_port_ref (in PortDef old_element)
        raises (Reflective::NotFound, Reflective::MofError) ;
    } ;
  } ;
```

FIG. 7.3 – *Projection de la classe* ComponentDef *en OMG IDL*

`Class` permet à la fois de créer des instances de type `ComponentDef`, mais aussi de retrouver, ou de détruire les instances qu'elle a au préalable créées. Les deux attributs donnent accès à une séquence de toutes les instances de `ComponentDef` créées par cette fabrique (`a`) ou contenues dans le référentiel (`b`). L'opération de destruction provient quant à elle de l'interface héritée `Reflective::RefObject`, qui fournit de plus des opérations génériques de manipulation.

L'interface `ComponentDef` fournit quant à elle une réification du concept de composant, ainsi qu'un ensemble de moyens pour manipuler ce concept. Un *ComponentDef* est en relation avec ses *PortDef* au travers d'une association et d'une référence. Des opérations sont donc générées pour ajouter, modifier, supprimer et retrouver les ports associés à un composant. Ces opérations réifient les capacités de navigation dans une architecture définie à l'aide de notre méta-modèle. Cette capacité de navigation à partir du composant est possible grâce à l'utilisation d'une référence. Sans référence, seule l'association aurait permis de faire des requêtes sur les ports d'un composant. Les attributs et opérations de la classe MOF sont définis en OMG IDL de manière standard. Nous ne nous attarderons donc pas sur leur projection.

Il est à noter que l'interface `ComponentDef` hérite de l'interface de sa fabrique. Ce choix est motivé dans le MOF par la possibilité d'interroger une instance afin de retrouver ses semblables et de pouvoir demander à une instance de créer une de ses semblables. Ceci ne représente pas l'opération de clonage, mais peut servir à sa mise en œuvre. Enfin, un type est défini pour manipuler des séquences de `ComponentDef`. Il est utilisé entre autres par la classe fabrique `ComponentDefClass`.

7.2.3 Projection en OMG IDL des associations MOF

Tout comme les classes, aux associations définies dans un méta-modèle est associé un schéma de projection vers le langage OMG IDL. Nous distinguerons ici deux types de projection des associations : les associations entre concepts définis au sein d'un même package, qu'ils soient hérités ou non ; et les associations entre un concept défini au sein du package et un concept importé d'un autre package.

7.2.3.1 Association *intra-package*

La figure 7.4 illustre la projection en OMG IDL de l'association *Provides* entre un composant et ses ports. Les définitions d'interfaces relatives à une association MOF se retrouvent, elles aussi, dans le module associé au package qui contient cette association dans le méta-modèle. Dans le cas de l'association *Provides*, les définitions provenant de la projection se retrouvent dans le module `ArchitecturalBase`.

La projection d'une association définit dans un premier temps une structure, `ProvidesLink`, qui réifie le lien entre une instance du référentiel décrivant un composant et une instance décrivant un port. Les champs de cette structure proviennent des noms donnés aux terminaisons de l'association *Provides*. La définition du type séquence permet la gestion des associations à cardinalité multiple. Ensuite, tout comme une classe, une association dispose d'une fabrique réifiée dans le référentiel. Le nom de l'interface associée porte le même nom que l'association et hérite de l'interface de base `Reflective::RefAssociation` fournissant un certain nombre d'opérations génériques pour la manipulation des associations. Cette fa-

```
module ArchitecturalBase {
  // ...
  struct ProvidesLink {
    ComponentDef owner ;
    PortDef port ;
  } ;

  typedef sequence<ProvidesLink> ProvidesLinkSet ;

  interface Provides : Reflective::RefAssociation {
    ProvidesLinkSet all_provides_links ()
      raises (Reflective::MofError) ;
    boolean exists (in ComponentDef owner, in PortDef port)
      raises (Reflective::MofError) ;
    ComponentDef owner (in PortDef port)
      raises (Reflective::MofError) ;
    PortDefSet port (in ComponentDef owner)
      raises (Reflective::MofError) ;
    void add (in ComponentDef owner, in PortDef port)
      raises (Reflective::MofError) ;
    void modify_owner (in ComponentDef owner, in PortDef port,
                       in ComponentDef new_owner)
        raises (Reflective::NotFound, Reflective::MofError) ;
    void modify_port (in PortDef port, in ComponentDef owner,
                      in PortDef new_port)
        raises (Reflective::NotFound, Reflective::MofError) ;
    void remove (in ComponentDef owner, in PortDef port)
        raises (Reflective::NotFound, Reflective::MofError) ;
  } ;
} ;
```

FIG. 7.4 – *Projection de l'association* Provides *en OMG IDL*

brique permet de connaître toutes les associations de type `Provides` définies au sein d'une architecture au travers de l'opération `all_provides_links ()`. Elle permet aussi de tester l'existance, de créer, de modifier, et de supprimer les associations de ce type. Les opérations `owner ()` et `port ()` permettent de rechercher le composant à qui appartient un port et tous les ports associés à un composant.

7.2.3.2 Association *extra-package*

Le cas des associations *extra-package* est peu différente du cas des associations *intra-package*. La projection d'une telle association définit une interface en tout point similaire. C'est sur la définition de la structure que des changement interviennent. Dans le cas de l'association *RunsOn* définie dans le package *Location* entre les concepts de composant (provenant du package architectural de base) et de site d'exécution (provenant du package de base du placement), la définition de la structure associée est illustrée dans la figure 7.5. Le premier changement est l'import du fichier contenant la définition du package `LocationBase`. Ensuite, la définition de la structure est construite simplement en préfixant le type « site d'exécution » par le nom du module le contenant : `LocationBase::HostDef`. Le reste de la définition est inchangée, puisque le type `ComponentDef` fait partie du module `Location` du fait de la relation d'héritage entre les packages. Les deux déclarations `typedef` permettent de faire connaître le type tel quel au sein du package `Location`.

```
#include <locationbase.idl>
#include <architecturalbase.idl>

module Location {
  interface PrimitiveDef : ArchitecturalBase::PrimitiveDef { } ;
  typedef sequence<PrimitiveDef> PrimitiveDefSet ;

  struct RunsOnLink {
    PrimitiveDef          comp ;
    LocationBase::HostDef host ;
  } ;

  typedef sequence<RunsOnLink> RunsOnLinkSet ;

  interface RunsOn : Reflective::Association {
    // ...
  } ;
  // ...
} ;
```

FIG. 7.5 – *Projection de l'association* RunsOn *en OMG IDL*

7.3 Mise en œuvre des référentiels

Cette section discute les différentes techniques de mise en œuvre des référentiels CODeX. Elle présente les choix d'implémentation et illustre le propos par leur expression en OMG IDLscript. Le choix d'utiliser un langage de script pour réaliser les référentiels CODeX est motivé par la simplicité d'utilisation de ce type d'environnement. En règle générale, l'implémentation d'un objet CORBA en OMG IDLscript est de deux à cinq fois plus petite que la même implémentation en Java. Ensuite, le langage IDLscript supporte l'héritage multiple (contrairement au langage Java), ce qui permet de respecter aisément la structuration du méta-modèle dans sa mise en œuvre.

L'utilisation d'un langage de script apporte une certaine souplesse par rapport aux langages compilés. Dans le contexte de ces derniers, toute modification impose une recompilation, et donc un redémarrage des services. Dans le cadre de notre projet, la possibilité de modifier dynamiquement le comportement des référentiels, par exemple dans le cas des actions architecturales, est un élément important. Enfin, l'aspect performance ne nous semble pas primordial pour ce qui est des interactions avec les référentiels.

Plusieurs choix d'implémentation peuvent être mis en œuvre. Certains de ces choix sont inspirés du travail de Xavier Blanc en relation avec la réalisation de l'outil de méta-modélisation M3J [10] et d'autres des projections réalisées par l'outil dMOF du DSTC [25].

7.3.1 Mise en œuvre des packages MOF dans le référentiel

Nous discutons ici de la mise en œuvre des interfaces OMG IDL présentées dans la section 7.2 à l'aide du langage OMG IDLscript. Dans le cadre de CODeX, trois types de mise en œuvre des packages sont utilisés : base architectural et base d'annotation ; annotation ainsi qu'intégration.

7.3.1.1 Mise en œuvre des packages de base

L'implémentation d'un package MOF est, dans le cadre de notre outil, relativement simple. Le package doit fournir une instance de fabrique pour chacun des concepts qu'il définit. Notre implémentation de package regroupe donc les instances de fabrique, soit la mise en œuvre des interfaces `*Class` présentées dans la section 7.2.2.

Les instances de fabriques sont gérées par des dictionnaires, un pour les classes (`_clss_ref`) et un pour les associations (`_asst_ref`) indexés par l'identifiant de ces concepts. Ces dictionnaires sont gérés par notre implantation de l'interface de base `reflective.RefPackage`. Celle-ci est principalement utilisée pour factoriser les implantations de packages, elle n'offre pas toutes les opérations liées à la reflexivité. Cette remarque est aussi vraie pour les implantations des interfaces `reflective.RefObject` et `reflective.RefAssociation`.

Le constructeur de la classe implantant un package de base (a) appel le constructeur de la classe `reflective.RefPackageImpl` (b), en précisant son identifiant et la référence du package le contenant (s'il y a lieu). Pour chaque classe (c) ou association (d) définie dans un package de base, une fabrique est créée et ajoutée dans le dictionnaire correspondant, avec son identifiant MOF comme clé. Pour chaque fabrique est définie une méthode accesseur, qui correspond à la lecture de l'attribut. Les accesseurs générés sont illustrés dans la figure 7.6

sur la partie relative à la gestion du concept de `ComponentDef` (e) et à l'association `Provides` (f).

```
   class ArchitecturalBasePackageImpl (reflective.RefPackageImpl) {

     idltype = ArchitecturalBase::ArchitecturalBasePackage ;

(a)  proc __ArchitecturalBasePackageImpl__ (self, id, ipkg) {
(b)    self.__RefPackageImpl__ (id, ipkg) ;
(c)    self._clss_ref ["p1class2"] =
          ComponentDefClassImpl ("p1class2", self) ;
(d)    self._asst_ref ["p1ass3"] = ProvidesImpl ("p1ass3", self) ;
       # ...
     }

(e)  proc _get_component_def_ref (self) {
       return self._clss_ref ["p1class2"] ;
     }
(f)  proc _get_provides_ref (self) {
       return self._asst_ref ["p1ass3"] ;
     }
     # ...
   }
```

FIG. 7.6 – *Implémentation du package* ArchitecturalBase *en OMG IDLscript (extrait)*

L'approche consistant à utiliser l'héritage de package implique la mise en œuvre de l'héritage multiple. Pour cette raison, il est important de bien maîtriser comment le langage d'implémentation gère ce type d'héritage. Dans le cas du langage OMG IDLscript, l'invocation des contructeurs des classes héritées est explicite. Une mise en œuvre simpliste risque d'initialiser plusieurs fois la classe `ArchitecturalBasePackageImpl` puisque celle-ci est héritée par tous les packages d'annotations. Pour éviter ce problème, l'utilisation d'un drapeau au sein des classes de base permet de savoir si le constructeur de cette classe a déjà été évalué ou non, et d'agir en conséquence.

7.3.1.2 Mise en œuvre des packages d'annotation

La mise en œuvre d'un package d'annotation est simple, comme l'illustre la figure 7.7. On ne trouve dans ce package que la définition relative à l'association `RunsOn`. La relation d'héritage avec la mise en œuvre du package de base architectural (a) fournit la définition de tous les concepts de ce package qui ne sont pas apparents. Le fait d'utiliser le package de base de placement n'apparaît pas ici, mais est visible dans la classe de mise en œuvre de l'association `RunsOnImpl` puisqu'elle contient une référence sur des instances de `HostDefImpl`, la mise en œuvre des sites d'exécution. La mise en œuvre du package d'annotation de placement ne fait donc que reprendre la définition du package architectural de base, dont il hérite, et ajoute à ses définitions l'association `RunsOn` dont la fabrique est instanciée et ajoutée au dictionnaire d'associations (c), et la méthode d'accès à l'attribut associé (d).

```
(a) class LocationPackageImpl (ArchitecturalBasePackageImpl) {

      idltype = LocationBase::LocationBasePackage ;

      proc __LocationPackageImpl__ (self, id, ipkg) {
(b)     self.__ArchitecturalBasePackageImpl__ (id, ipkg) ;
(c)     self._asst_ref ["p4ass1"] = RunsOnImpl ("p4ass1", self) ;
      }

(d)   proc _get_runs_on_ref (self) {
        return self._asst_ref ["p4ass1"] ;
      }
    }
```

FIG. 7.7 – *Implémentation du package d'annotation* Location *en OMG IDLscript*

L'initialisation de l'objet représentant le package de base (b) est conditionnelle. Si un autre package d'annotation a déjà initialisé celui-ci, alors cette invocation (b) ne réalisera aucun traitement. Dans tous les cas, il faut que l'initialisation des associations définies au sein de ce package soient évaluée (c).

7.3.1.3 Mise en œuvre du package d'intégration

Le méta-modèle définit simplement l'intégration des packages d'annotation ; il en va de même pour sa mise en œuvre. Simple comme celle des packages d'annotations, elle est illustrée par la figure 7.8. L'implémentation du package `MyADL` hérite donc de tous les packages d'annotation (a, b et c), ce qui se traduit dans le cas présent par un héritage de l'implémentation des classes `LocationPackageImpl`, `LocationPackageImpl` et `DynamismPackageImpl`. La phase d'initialisation de `MyADLPackageImpl` n'a comme activité que l'initialisation des classes héritées (d, e et f). Comme aucun concept n'est défini au niveau de l'intégration dans notre méta-modèle, aucune définition ne vient s'ajouter aux définitions existantes.

```
(a) class MyADLPackageImpl (LocationPackageImpl,
(b)                         ImplementationPackageImpl,
(c)                         DynamismPackageImpl) {

      idltype = MyADL::MyADLPackage ;

      proc __MyADLPackageImpl__ (self, id, ipkg) {
(d)     self.__LocationPackageImpl__ (id, ipkg) ;
(e)     self.__ImplementationPackageImpl__ (id, ipkg) ;
(f)     self.__DynamismPackageImpl__ (id, ipkg) ;
      }
    }
```

FIG. 7.8 – *Implémentation du package d'intégration* MyADL *en OMG IDLscript*

La figure 7.9 illustre aussi la gestion de l'héritage multiple dans la mise en œuvre du

package d'intégration, et l'utilisation du drapeau d'initialisation. Sans cette précaution, seule l'initialisation du dernier package d'annotation serait effective puisqu'elle écraserait les initialisations précédentes liées aux autres packages d'annotation : l'initialisation des dictionnaires serait reprise à chaque fois à zéro. Une flèche pleine indique une initialisation transitive. Une flèche en pointillés indique une initialisation non transitive. Le coût lié à la gestion de l'héritage multiple est minime, alors que son intérêt est important : production simple des différents packages en respectant le découpage et l'approche suivis dans la définition du méta-modèle.

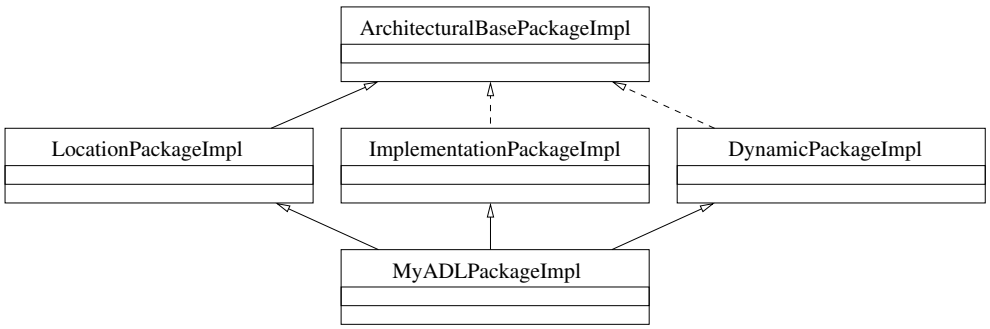


FIG. 7.9 – *Gestion de l'héritage multiple lors de l'initialisation des classes de packages*

7.3.2 Mise en œuvre des classes MOF dans le référentiel

Une classe MOF est projetée en deux interfaces OMG IDL (voir section 7.2.2). L'implémentation d'une classe MOF découle donc dans la mise en œuvre de ces deux interfaces. La classe `ComponentDefClassImpl` met en œuvre la fabrique référencée par le package `ArchitecturalBase` servant à créer, rechercher et supprimer la représentation des instances de composants de l'architecture. La classe `ComponentDefImpl` est quant à elle la mise en œuvre de cette représentation.

La figure 7.10 présente un extrait de cette mise en œuvre. L'interface `ComponentDef` hérite de l'interface `ComponentDefClass`, mais les implantations sont dissociées. L'implantation `ComponentDefImpl` hérite de `reflective.RefObjectImpl` (a) qui représente la classe de base pour les classes et instances MOF. L'utilisation de la variable `_proxy` (b) permet à une instance de `ComponentDefImpl` de connaître la référence de sa fabrique. Les opération héritées de l'interface de cette fabrique (f) sont implantés par délégation.

La mise en œuvre de la classe `ComponentDefImpl` se décompose en deux parties comme pour les packages. Le constructeur de cette classe regroupe la définition des structures de données. Comme une instance de composant est en relation avec ses ports au travers d'une association, sa mise en œuvre contient un tableau des représentants de ports en relation avec cette instance (c). Un second attribut définit quant à lui les caractéristiques du port [1] comme sa cardinalité (d). Associés à ce tableau, les opérations d'ajout (e), de modification et de suppression de ports utilisées pendant la phase de définition d'une instance de composant sont réalisées. Une dernière opération pour accéder à tous les ports d'un composant pendant la phase d'exploitation de l'architecture est définie. La mise en œuvre des classes MOF réifie

1. Ces caractéristiques proviennent de l'association entre un composant et un port.

```
(a) class ComponentDefImpl (reflective.RefObjectImpl) {

      idltype = ArchitecturalBase::ComponentDef ;

      proc __ComponentDefImpl__ (self, id, ipkg, proxy) {
        self.__RefObjectImpl__ (id, ipkg) ;
(b)     self._proxy = proxy ;
(c)     self._port_ref = [] ;
(d)     self._port_ref_desc = { "lower": "0", "upper": "*",
               "isOrdered": "false", "isUnique": "false" } ;
      }

(e)   proc add_port_ref (self, new_element) {
        self._port_ref.append (new_element) ;
      }

(f)   proc all_of_type_component_def (self) {
        self._proxy.all_of_type_component_def () ;
      }
      # ...
    }
```

FIG. 7.10 – *Implémentation de la classe* `ComponentDef` *du package* `ArchitecturalBase` *en OMG IDLscript*

aussi les attributs et les opérations des classes définies dans le méta-modèle. Ceux-ci sont implémentés comme tout attribut ou opération OMG IDL classique.

Dans le cas des packages d'annotation, cette mise en œuvre n'est pas remise en cause du fait de la relation d'héritage au niveau des packages. La réalisation des concepts définis sous forme de classes dans le plan de base architectural sont repris tels quels. L'implémentation de `ComponentDef` est donc partagée par le plan de base et par tous les plans d'annotations. Il en va de même pour le plan d'intégration. Ceci est possible grâce à l'approche suivie au moment de la définition du méta-modèle. Comme la définition des associations au niveau d'annotation ne doivent pas être associées à la définition de références sur les concepts provenant du plan de base, ces dernier restent inchangés (aussi bien au niveau du méta-modèle qu'au niveau de la projection en OMG IDL).

7.3.3 Mise en œuvre des associations MOF dans le référentiel

La mise en œuvre des associations MOF est similaire à la mise en œuvre des classes MOF en termes de structuration. La figure 7.11 présente un extrait de l'implémentation de l'association `Provides`. Cette mise en œuvre est aussi valable dans le cas des associations définies au sein des packages d'annotation.

L'implantation d'une association MOF hérite de la classe de base `reflective.RefAssociationImpl` (a). Cette relation d'héritage précise non seulement la relation d'héritage au niveau des interfaces, mais permet aussi la factorisation de code – comme c'est le cas pour les packages et les classes. L'invocation du constructeur de base (b) permet l'initialisation des structures

de données comme la gestions des structures IDL réifiant les associations entre un composant et un port (voir section 7.2.3).

Les opérations associées à la gestion de ces structures de données permettent d'ajouter une relation entre un composant et un de ses ports (d), de modifier, de supprimer pendant la phase de conception d'une architecture, ainsi que de retrouver tous les ports d'une instance de composant (d) ou l'instance de composant à qui appartient un port donné. Sans la définition de référence dans le concept `ComponentDef`, l'association représenterait le seul moyen de naviguer au sein d'une architecture.

```
(a) class ProvidesImpl (reflective.RefAssociationImpl) {

      idltype = ArchitecturalBase::Provides ;

      proc __ProvidesImpl__ (self, id, ipkg) {
(b)     self.__RefAssociationImpl__ (id, ipkg) ;
      }

(c)   proc add (self, owner, port) {
        self._links.append ([owner, port]) ;
      }

(d)   proc port (self, owner) {
        res = [] ;
        for lnk in self._links { ;
          if (lnk[0].is_equivalent (owner)) res.append (lnk[1]) ;
        }
        return res ;
      }
      # ...
    }
```

FIG. 7.11 – *Implémentation de la classe d'association* `Provides` *du package* `ArchitecturalBase` *en OMG IDLscript (extrait)*

Cette mise en œuvre n'est instanciée qu'une seule fois au sein de la représentation d'une architecture, et partagée. Cette instance gère l'ensemble des associations de type `Provides`, quelque soit leur nombre. Cette mise en œuvre suit scrupuleusement la spécification du MOF. Toutefois, dans le cas d'architectures de taille importante, ce choix est discutable. En effet, une architecture regroupant un grand nombre d'instances de composants regroupe aussi un grand nombre d'associations `Provides` (au moins autant que d'instances de composants, mais plutôt n fois supérieur –n étant le nombre moyen de ports par instance de composant). Le MOF impose donc une vision centralisée sur ce point. L'utilisation de référentiels fédérés est une solution à envisager dans le cas d'architectures importantes et ce point vient en renforcer la nécessité.

7.3.4 Mise en œuvre des actions architecturales

Comme nous l'avons discuté dans le § 6.3.3.2, les actions architecturales définissant la dynamique des architectures sont représentées sous la forme de classes MOF. La mise en œuvre des actions est donc similaire à celle des autres classes MOF (voir section 7.3.2). Dans le cas des actions architecturales, les traitements relatifs à la dynamique sont définis au sein de l'opération `eval ()`. Nous avons défini dans notre méta-modèle les actions comme des séquences d'opérations élémentaires (a) sur les instances de composants ou connecteurs – au travers de l'association *IsDefinedAs*.

La mise en œuvre par défaut d'une action architecturale est donc relativement simple : les différentes opérations de base sont invoquées successivement. Toutefois, cette mise en œuvre ne peut être automatisée (b) car la sémantique du `eval ()` ne peut être exprimée dans le méta-modèle.

```
    class ActionDefImpl (reflective.RefObjectImpl) {

      idltype = DynamismBase::ActionDef ;

      proc __ActionDefImpl__ (self, id, ipkg) {
        self.__RefObjectImpl__ (id, ipkg) ;
(a)     self._op_base_def_ref = [] ;
        # ...
      }

(b)   proc eval (self) {
        # méthode implanté manuellement
        for op in self._ops {
          op.eval () ;
        }
      }
      # ...
    }
```

FIG. 7.12 – *Implémentation de la classe `ActionDef` du package `DynamismBase` en OMG IDLscript (extrait)*

La figure 7.12 illustre la mise en œuvre par défaut de la classe MOF `ActionDef`. Dans ce cas, les opérations de base sont invoquées séquentiellement. Pour optimiser la mise en œuvre du déploiement, l'invocation en parallèle de certaines opérations est une solution intéressante. Pour cela, l'approche la plus appropriée n'est pas uniquement de modifier la mise en œuvre de l'opération `eval ()` mais d'introduire au niveau du méta-modèle la notion d'opération parallélisable. Il devient ainsi possible de définir des groupes d'opérations de base, soit séquentielles, soit parallélisables. Leur mise en œuvre devient alors automatisable. Dans le cadre de nos expérimentations actuelles nous n'avons mis en œuvre que la version séquentielle des groupes d'opérations de base.

7.4 Conclusion

Après avoir défini le méta-modèle d'un ADL, la production de son environnement associé à l'aide de CODeX représente la phase de transition vers l'utilisation de cet ADL. Ce chapitre présente les techniques mises en œuvre pour réaliser cette production de manière automatique. L'ensemble des projections vers le langage OMG IDL et la mise en œuvre des référentiels sont réalisés à l'aide de l'outil *CODeX M2 tool*. Cet outil exploite la représentation d'un méta-modèle en XMI pour générer les interfaces OMG IDL et les implémentations en OMG IDLscript des référentiels associés aux ADLs.

Les référentiels, tels que nous les avons présentés ici, sont destinés à définir les modèles abstraits des architectures, les PIMs. En effet, les projections en OMG IDL des interfaces de référentiels et leur mise en œuvre en OMG IDLscript sont, à ce point, indépendants des technologies. Les référentiels ainsi produits sont toutefois utilisables pour définir les PIMs. Il est donc possible de définir une première version des architectures qui sera par la suite à charger dans un référentiel dédié à une technologie en vue de sa finalisation et de son exploitation.

Nous avons pu voir tout au long de ce chapitre, que la mise en œuvre d'un méta-modèle est une activité automatisable et relativement abordable, même dans le contexte d'un usage particulier. **La mise en œuvre des référentiels suit, tout comme la définition des méta-modèles, l'approche par séparation des préoccupations.** Ceci est possible grâce à la disponibilité de l'héritage multiple dans le langage d'implémentation choisi. Cette mise en œuvre pourrait aussi se faire avec un langage comme Java, donc ne supportant pas ce type d'héritage, mais elle serait plus complexe. Le résultat serait de produire davantage de code. Soit, l'implémentation d'une classe contiendrait l'intégralité de l'implémentation des classes dont elle hérite, soit un mécanisme de délégation serait mis en œuvre.

Le choix du langage OMG IDLscript est donc avantageux car il apporte une simplicité de mise en œuvre des référentiels. Il apporte aussi, et c'est la principale motivation de notre choix, de la flexibilité dans l'utilisation des référentiels. Il est ainsi possible de modifier le comportement des référentiels sans repasser par la phase de projection des méta-modèles. Enfin, l'utilisation de la version *jIdlscript*, qui est une implémentation en Java de la spécification OMG IDLscript, apporte un dernier intérêt non négligeable. *jIdlscript* permet aussi bien de manipuler des objets CORBA que des objets Java. Les implémentations de référentiels sont ainsi utilisables pour produire des applications avec les technologies *Java RMI* et *Enterprise Java Beans*

Sur un plan plus technique, une petite comparaison a été réalisée entre les référentiels produits par *CODeX M2 tool* et l'outil *M3J* en terme de nombre de lignes de code. Si l'on ne considère que le plan de base architectural, du fait de la limitation de M3J à un package par modèle, M3J produit moins de 10 000 lignes de Java contre moins de 2 000 lignes d'OMG IDLscript dans le cas de *CODeX M2 tool* pour des fonctionnalités équivalentes. Même si ce code est généré, il est intéressant de produire des implémentations simples. Par extrapolation, M3J produirait pour le méta-modèle complet que nous avons défini environ 60,000 lignes de code contre moins de 5,000 lignes pour notre outil. Ces comparaisons ne sont pas à prendre en termes d'approche, qui est très similaire, mais en terme de langage d'implémentation utilisé. Elles représentent un argument en faveur du langage OMG IDLscript sur le plan de

la simplicité des référentiels produits.

Le référentiel que nous avons présenté ici doit maintenant être spécialisé pour être totalement utilisable avec une technologie donnée. C'est la transition PIM vers PSM. Cette transformation vise à fournir une version des traitements contenus dans le référentiel adaptée pour le modèle technologique visé. Notre première expérimentation s'est déroulée dans le contexte du *CORBA Component Model* (CCM). Nous discutons dans le prochain chapitre de l'utilisation de ce référentiel dans le cadre du CCM, en présentant tout d'abord sa spécialisation puis son utilisation pour définir une application et mettre en œuvre le déploiement.

Chapitre 8

Mise en œuvre de CODeX dans le cadre du CCM

Les chapitres précédents ont présenté notre stratégie de définition du méta-modèle d'un ADL et l'environnement de manipulation d'architectures associé. Ce chapitre présente comment cet ADL et son environnement sont spécialisés pour un modèle technologique de composants. Cette spécialisation regroupe à la fois l'introduction des concepts du modèle de composants technologique dans le méta-modèle, la modification du référentiel pour son utilisation avec le modèle de composants technologiques et enfin la projection des architectures définies dans le référentiel vers ce modèle de composants pour réaliser la mise en œuvre des applications.

- La section 8.2 présente la projection du méta-modèle défini dans le chapitre précédent vers le modèle de composants CORBA qui sert de base à nos expérimentations.
- La section 8.3 présente la projection des définitions de composants et connecteurs d'une architecture vers le langage OMG IDL3 afin de mettre en œuvre une application.
- La section 8.4 présente la mise en œuvre des opérations architecturales élémentaires dans le contexte du modèle de composants CORBA.
- La section 8.5 illustre l'utilisation de l'environnement de manipulation d'architectures ainsi défini et spécialisé pour le modèle de composants CORBA.

8.1 Introduction

Ce chapitre présente la spécialisation du référentiel de base que nous avons défini pour le *CORBA Component Model* (CCM). La discussion porte ici à la fois sur le niveau de définition des architectures (M1) et sur le niveau de représentation des applications (M0) (cf section 5.3.2). C'est principalement ce dernier niveau qui est spécialisé pour le CCM. Il représente la transition entre la représentation d'une architecture et une instance d'application, le « lien » permettant le passage d'un niveau à l'autre. C'est aussi au niveau de la représentation des applications que le processus de déploiement est mis en œuvre pour le modèle technologique cible. La figure 8.1 illustre l'organisation des niveaux M1, M0 et applicatifs. La mise en œuvre de cette organisation peut prendre plusieurs formes. Le niveau M0 peut co-exister dans le même référentiel que le niveau M1 (approche que nous avons mise en œuvre) ou exister dans un référentiel dédié. Une définition d'architecture (niveau M1) est instanciée pour produire une ou plusieurs représentations d'applications (niveau M0) correspondant à une ou plusieurs instances d'applications.

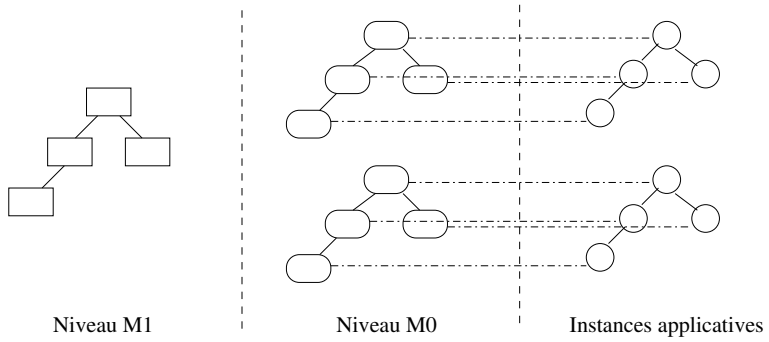


FIG. 8.1 – *Organisation de la représentation des applications*

En addition de la spécialisation du niveau de représentation du référentiel, les règles de projection des architectures vers le langage OMG IDL 3 sont aussi présentées. Ces règles permettent de fournir les interfaces pour le CCM des types de composants manipulés au sein de l'architecture afin de développer les différents composants de l'application. Ces interfaces représentent la base du travail des développeurs et des intégrateurs pour mettre à disposition les implémentations de composants qui seront utilisées pendant la phase de déploiement des applications.

Enfin, nous illustrons l'utilisation de notre environnement pour définir l'architecture de l'application classique « le dîner des philosophes ». Une fois cette architecture définie, nous utilisons l'environnement CODeX pour l'exploiter afin de produire les interfaces OMG IDL 3 et leur implémentation. Enfin, les traitements associés au déploiement, et contenus dans le référentiel, seront utilisés pour réaliser le déploiement de cette application.

8.2 Environnement pour le *CORBA Component Model*

L'environnement de manipulation d'architectures pour le *CORBA Component Model* (CCM) regroupe plusieurs extensions au référentiel tel que nous l'avons présenté dans le chapitre 7. Tout d'abord, le référentiel est spécialisé pour le CCM. Cette spécialisation est le lien entre la définition d'une architecture et les instances d'applications qui en résultent. Cette section définit le niveau M0 de la pile de méta-modélisation de CODeX. Ce niveau est utilisé lors de la phase de déploiement d'une application pour instancier une représentation de l'application qui est déployée. Ce niveau peut co-exister au sein du référentiel défini dans le chapitre précédent, ou être déporté dans un référentiel auxiliaire.

8.2.1 Spécialisation du référentiel

La spécialisation du référentiel a pour objectif de faire le lien entre les descriptions d'architectures et les instances des applications ainsi définies. Cette spécialisation représente la fourniture d'un ADL pour un modèle de composants particulier. Dans le contexte de nos expérimentations, notre ADL devient alors *MyADLCCM*. Ici encore, nous utilisons l'héritage de package au sens du MOF pour produire ce plan global de l'ADL pour le CCM. En addition

de cette relation entre packages, certains des concepts, comme *primitif*, *port* ou *composite* seront étendus de manière individuelle afin d'être spécialisés pour le CCM. Cette spécialisation définit un ensemble de patrons, les squelettes des classes de mise en œuvre, pour définir les traitements relatifs à la dynamique dans le contexte du modèle technologique choisi. Il ne reste alors plus qu'à implémenter ces traitements, dans le cas présent pour le CCM.

8.2.1.1 Spécialisation des interfaces du référentiel

La production des interfaces du référentiel étendu passe par la définition de règles de projection. Pour chaque concept défini dans le méta modèle une règle doit être définie. Nous présenterons ici uniquement les règles de projection pour les composants et les composites, toutes les autres suivent la même approche. Ces règles de projection représentent la transformation du PIM (*Platform Independent Model*) d'une architecture vers un PSM (*Platform Specific Model*). Cette transformation représente le passage du niveau M1 –la définition des architectures– au niveau M0 –la représentation des instances d'applications. Pour une même définition d'architecture il peut exister plusieurs instances d'application. Par exemple, si l'architecture d'un agenda est définie, plusieurs instances de cet agenda peuvent exister au niveau applicatif. Pour chacun de ces agendas, une représentation existe au niveau M0. C'est dans la phase de déploiement d'une application que ces instances de niveau M0 seront créées parallèlement aux instances applicatives.

Spécialisation d'un composant primitif Cette règle de projection est représentative de tout concept du méta-modèle existant, plus ou moins directement, dans le modèle technologique. L'interface d'une instance de composant au niveau du PSM (pour le CCM) est très similaire à l'interface du niveau PIM. A ce niveau, une instance de composant fournit toujours, par exemple, un certain nombre de ports et dispose d'un nom. Cette représentation est complétée par la connaissance de l'instance applicative qu'elle désigne.

La figure 8.2 présente la définition du concept de composant pour le CCM. Cette définition reprend donc toutes les définitions du niveau PIM, c'est-à-dire qu'elle hérite de la définition de `ComponentDef`. Cette relation explicite bien le fait que la représentation d'une instance pour le CCM est le même concept que la représentation d'une instance au niveau de l'architecture. En addition de ces définitions, `ComponentDefCCM` hérite de `CCMObjectHolder`. Cette seconde relation d'héritage définit la spécialisation de la représentation du concept de composant primitif pour le CCM. Cette spécialisation fournit le lien entre la représentation de l'architecture et les instances applicatives. `CCMObjectHolder` « contient » la référence de l'instance de composant CORBA qui met en œuvre un composant de l'architecture. Cette classe contient un unique attribut de type référence de composant CORBA.

La figure 8.3 illustre la projection en interface OMG IDL de cette règle de spécialisation. Cette interface dédiée à une utilisation dans le contexte du CCM regroupe tous les ports de l'instance de composant, par la relation d'héritage. Toutefois, dans son utilisation, ces ports ne sont plus les ports de l'architecture, mais bien les ports de l'instance applicative. L'interface `PortDef` est elle aussi spécialisée pour le CCM, et les ports connus par une instance de `ComponentDefCCM` sont des instances de port de type `PortDefCCM`. Un port spécialisé contient quant à lui la référence du port « physique » du composant CCM associée. Il n'est toutefois

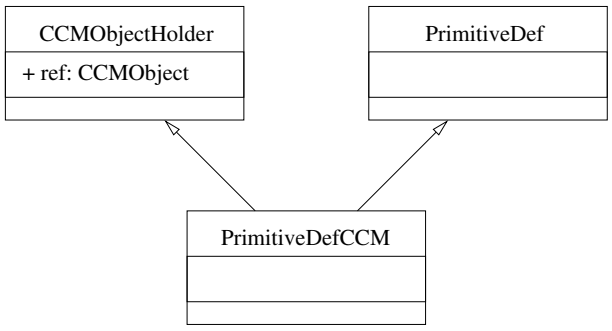


FIG. 8.2 – *Spécialisation du concept de composant primitif pour le CCM*

pas nécessaire de redéfinir toutes les opérations de manipulation de ports, un `PortDefCCM` est un `PortDef` et donc manipulable comme tel.

```
interface CCMObjectHolder {
  attribute Components::CCMObject ref ;
} ;

interface PrimitiveDefCCM : PrimitiveDef, CCMObjectHolder {
} ;
```

FIG. 8.3 – *Spécialisation de l'interface `PrimitiveDef` pour le CCM*

Lors de la phase de déploiement, pour toute instance de composant primitif définie dans l'architecture, une instance de `PrimitiveDefCCM` sera créée en même temps que l'instance applicative correspondante. C'est à ce moment que sera fixée la valeur de l'attribut `ref`. Lors de la suppression d'une instance de composant de l'application, cette instance de représentation sera elle aussi supprimée. Ainsi, il ne doit jamais y avoir dans le référentiel d'instance de représentation avec un attribut `ref` non fixé, si ce n'est dans une phase transitoire.

Spécialisation d'un composite Le CCM ne dispose pas du concept de composite. De plus, les instances de composants CCM sont nécessairement co-localisées avec leurs ports ; elles ne peuvent donc pas être utilisées pour mettre directement en œuvre les composites. Cette règle de projection est représentative de tout concept du méta-modèle n'existant pas dans le modèle technologique visé. Nous discutons dans la section 8.3.1 de la projection d'une architecture en OMG IDL 3. Une des règles définit qu'un composite est projeté en un ensemble de composants (au sens du CCM), chacun d'entre eux représentant un port du composite. Dans la mise en œuvre pour le CCM il apparaît donc qu'un port de composite n'est pas similaire à un port de composant. C'est pourquoi la règle de projection des composites est un peu différente.

Tout comme un composant, un composite contient une liste de ses ports associés. Nous avons vu dans le cas de la projection des composants que la représentation d'un port pour le CCM contient la référence du port effectif. Dans le cas des composites, les ports sont définis

comme des composants CCM, il est donc nécessaire pour le composite, non seulement de connaître les références de ses ports, mais aussi des composants qui les implantent. Dans le CCM, il existe deux notions de références : les références de base, qui identifient une instance de composant, et les références de facettes, qui identifient une facette précise d'une instance de composant. Afin d'unifier l'utilisation, le composite doit connaître la référence de facette, pour se comporter comme une instance de composant, et la référence de base, pour pouvoir manipuler ses propres ports.

La figure 8.4 illustre la spécialisation des composites de notre méta-modèle. La définition `CompositePort` (qui doit être perçu comme un type de données) associe une référence de facette avec le nom du port. Cette définition est utilisée par un composite (attribut `crefs`) pour stocker la liste de ses ports. Pour connaître à la fois les ports et les composants les représentant, un `CompositeDefCCM` dispose aussi d'un attribut `prefs` référençant ces composants. Ces deux attributs sont gérés de manière synchronisée.

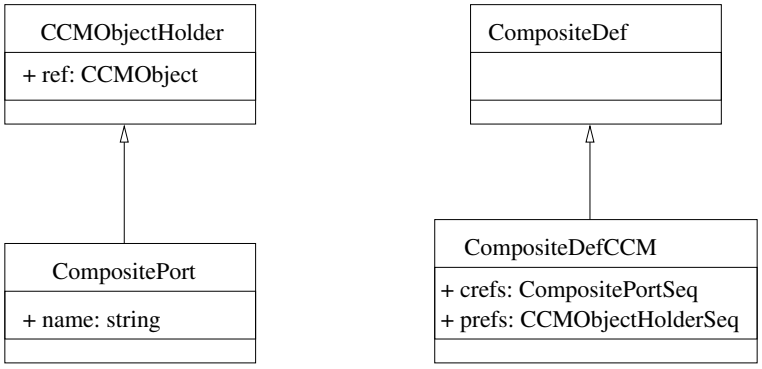


FIG. 8.4 – *Spécialisation du concept de composite pour le CCM*

La projection en interfaces OMG IDL de cette définition est illustrée dans la figure 8.5. Cette projection est un peu particulière du fait de l'utilisation de `CompositePort` comme un type de données. Cela explique l'utilisation d'une structure OMG IDL pour représenter les ports de composites.

```
interface CompositePort : CCMObjectHolder {
  attribute string                name ;
  attribute Components::CCMObject ref ;
} ;
typedef sequence<CompositePort> CompositePortSeq ;

interface CompositeDefCCM : MyADL::CompositeDef {
  readonly attribut CompositePortSeq crefs ;
  readonly attribut CCMObjectSeq     prefs ;
} ;
```

FIG. 8.5 – *Projection de l'interface `CompositeDef` pour le CCM*

De l'utilisation de l'héritage Pour les deux projections présentées ici, comme pour toutes les autres, le choix d'utiliser une relation d'héritage et non de définir de nouvelles interfaces n'est pas uniquement le résultat d'une volonté d'avoir des règles de projection simples. Premièrement, une certaine dynamicité doit exister à ce niveau pour la mise en œuvre des actions architecturales. Il est nécessaire de pouvoir supprimer une instance de composant d'un composite (pendant les phases de reconfiguration par exemple). Ensuite, bien que le CCM ne permette pas de définir dynamiquement de nouveaux ports sur une instance de composant, il peut être intéressant de pouvoir le faire au moins sur les composites, qui sont des composants. De plus, il peut être nécessaire de supprimer l'accès à certains ports d'une instance de composant, sans pour autant supprimer cette instance de l'application, par exemple pour la suspension temporaire d'un service. Dans ce cas, les opérations de suppression de ports sont importantes aussi sur les instances de composants.

8.2.1.2 Mise en œuvre des extensions du référentiel

De même que la mise en œuvre du référentiel indépendant des plates-formes, la mise en œuvre des extensions pour le CCM est simple. Nous n'allons présenter ici que la mise en œuvre de l'interface `PrimitiveDefCCM` qui est illustrée dans la figure 8.6. Pour tous les concepts du plan d'intégration de notre ADL qui sont étendus dans le plan dédié au CCM il est nécessaire de produire une implémentation. Dans le cas de `PrimitiveDef`, l'unique extension réside dans l'attribut contenant la référence de l'instance de composant du niveau applicatif, représentée par l'instance de `PrimitiveDefCCM` dans le référentiel. Cette instance de la représentation de l'architecture permet le « passage » entre la représentation et l'application elle-même. Ce « passage » donne accès à une instance de composant de l'application pour, par exemple, connaître son état. C'est aussi au travers de ce « passage » que la représentation de l'application peut agir sur celle-ci, par exemple pour supprimer l'instance de composant du niveau applicatif.

8.2.1.3 Production de la spécialisation du référentiel

Tout comme la partie abstraite du référentiel correspondant au PIM de notre ADL, la partie concrète, le PSM pour le CCM, est produite à partir de l'expression de la spécialisation du méta-modèle de l'ADL en XMI. De la même manière, la mise en œuvre de ce référentiel spécialisé devrait être produite automatiquement. Actuellement, les règles de projection sont implantées au niveau de deux générateurs spécialisés pour le CCM : un générateur produit les interfaces OMG IDL et l'autre des implantations en IDLscript de ce référentiel.

En addition des concepts du méta-modèle spécialisés pour le CCM, ces générateurs produisent l'implémentation spécialisée du package d'intégration, c'est-à-dire l'ensemble des fabriques des instances du référentiel dans leur version CCM. Ces fabriques sont utilisées pendant la phase de déploiement pour créer les instances du niveau M0 de notre pile de méta-modélisation. Ce sont les méta-instances effectives des instances applicatives.

```
class PrimitiveDefCCMImpl (PrimitiveDefImpl) {

  idltype = MyADL_CCM::PrimitiveDefCCM ;

  proc __PrimitiveDefCCMImpl__ (self, meta) {
    self.__PrimitiveDefImpl__ (0, 0, CORBA.Object._nil) ;
    self._meta = meta ;
    self._ref  = CORBA.Object._nil ;
  }

  proc _get_ref (self) {
    return self._ref ;
  }

  proc _set_ref (self, ref) {
    self._ref = ref ;
  }
}
```

FIG. 8.6 – *Mise en œuvre de l'interface* `PrimitiveDefCCM` *en IDLscript*

8.3 Support à la mise en œuvre des applications

Idéalement, n *traducteur* devrait produire les interfaces OMG IDL 3 nécessaires à la mise en œuvre de cette architecture dans le cadre du CCM. Ce *traducteur* projeterait la définition abstraite de l'architecture vers une définition concrète : il représente la transformation « PIM vers PSM » de la MDA vis-à-vis de l'architecture. Actuellement, cette projection n'est pas automatisée. Les règles de projection présentées ici ont été définies et utilisées manuellement.

Afin de faciliter la transition entre définition de l'architecture des applications et mise en œuvre de celles-ci, il est nécessaire de définir une transformation. Cette transformation est un ensemble de règles de projection définissant les interfaces des éléments de l'application dans un modèle technologique donné. Ce point place notre travail dans le cadre des langages de configuration tels que nous les avons présentés dans la section 3.4.

L'idée sous-jacente n'est pas uniquement de disposer d'une version de l'architecture mais de l'utiliser pour produire des applications, comme le fait Olan par exemple. Dans le cadre de nos expérimentations, les règles de projection définissent les interfaces OMG IDL 3 des applications. La définition de ces règles est réalisée en commun par le concepteur de l'ADL, qui maîtrise les concepts de cet ADL, et un spécialiste du modèle technologique cible. Ils vont tout deux définir les règles de correspondance entre les concepts de l'ADL et les concepts disponibles dans le modèle technologique.

8.3.1 Règles de projection vers le langage OMG IDL 3

Les actions ne sont pas discutées ici puisqu'elles ne sont réifiées que dans le référentiel et ne sont pas implémentées au niveau applicatif. Il n'y a donc pas de projection des représentations des actions vers le langage OMG IDL 3. Nous discutons de la mise en œuvre des actions

architecturales pour le CCM dans la section 8.4.

8.3.1.1 Projection des composants primitifs

Le concept de composant est similaire dans le CCM et dans notre méta-modèle : c'est une entité de traitement offrant des ports (voir section 2.4.2). La projection des primitifs des architectures est donc relativement directe. Dans le cas du dîner des philosophes, un philosophe est défini comme une instance de composant disposant de deux ports de type `OutputPortDef` représentant ses deux mains et destinés à utiliser des ports d'utilisation des fourchettes (voir section 8.5.2). Ces deux ports sont définis comme synchrones. De plus, un philosophe dispose d'un attribut représentant son nom (`name`). La figure 8.7 illustre la projection d'un philosophe en OMG IDL 3.

```
component Philosophe {
  uses IFourchette main_droite ;
  uses IFourchette main_gauche ;
  attribute string name ;
} ;
```

FIG. 8.7 – *Projection de la définition d'un philosophe en OMG IDL 3*

De manière plus générale, une instance de composant de l'architecture est projetée en une interface OMG IDL 3 de définition d'un composant, avec l'utilisation du mot clé `component`. Les ports de type *input* de l'architecture sont projetés en des facettes de composant (mot clé `provides`) pour les ports synchrones, et en des puits d'événements (mot clé `consumes`) pour les ports asynchrones. Pour les ports de type *output*, les règles suivent le même principe : les ports synchrones sont projetés en réceptacles (mot clé `uses`). Les propriétés des composants sont quant à eux directement projetés vers des attributs OMG IDL 3 (mot clé `attribute`).

Nos expérimentations ont uniquement mis en œuvre les ports synchrones. Dans le cas de ports asynchrones, la projection devrait définir des sources d'événements (mot clé `emits` pour un connecteur de type $1 - vers - 1$ et `publishes` pour un connecteur de type $1 - vers - n$). Toutefois, dans le modèle de composant CORBA les ports reflétant des sources d'événements ne sont pas définis à l'aide d'interfaces offrant des opérations, mais à l'aide de *valuetype* représentant des types d'événements. Pour utiliser des événements il serait nécessaire d'étendre la définition de notre méta-modèle pour associer les ports de sortie asynchrones d'un composant avec le type d'événement émis. La différentiation entre spécification d'événement ou d'opérations pourrait alors être exprimé à l'aide d'une contrainte OCL liée au mode (synchrone ou asynchrone) du port.

8.3.1.2 Projection des connecteurs

De par sa définition, le CCM ne fournit pas le concept de connecteur. Il en résulte que la projection des connecteurs du méta-modèle peut prendre deux formes. La première, la plus simple, prend en charge les connecteurs sans traitement associé. Un tel connecteur est simplement un lien entre deux composants. Dans ce cas, il n'y a pas de projection en OMG

IDL 3. La représentation de ce type de connecteurs dans le référentiel va uniquement réifier le lien logique entre les ports de deux instances de composants.

La seconde forme prend en charge les connecteurs ayant des traitements associés. C'est le cas des connecteurs sur lesquelles on fixera, par exemple, une stratégie de « log » des invocations. Dans ce cas, la projection définit un nouveau type de composant CCM qui offre et utilise deux ports de même type que les ports des instances de composants mis en relation par le connecteur.

Dans le cas où deux instances de composants A et B sont en relation au travers de ports de type *Service* en mode synchrone, si l'on souhaite que le connecteur entre ces deux instances fasse un « log » de toutes les invocations, alors un nouveau type de composant CCM doit être défini : *CLog*. La figure 8.8 illustre la définition de ce nouveau type de composant CCM pour prendre en charge la propriété de « log ». Toute invocation d'une opération émise par une instance de A sera reçue par l'instance de *Clog*, ajoutée dans une base de « logs » et retransmise à l'instance de B. La réponse suivra le chemin inverse.

```
component Clog {
  provides Service ref ;
  uses     Service use ;
} ;
```

FIG. 8.8 – *Projection d'un connecteur incluant des traitements en OMG IDL 3*

Les traitements relatifs à l'établissement et à la suppression des connecteurs sont, quant à eux, contenus dans le référentiel et ne sont pas réifiés au niveau applicatif. Cette séparation se justifie par le fait que les traitements sont d'ordres architecturaux, donc non fonctionnels vis-à-vis de l'applicatif. Cette approche permet de mettre en œuvre la dynamique des architectures de manière indépendante des applications, et donc de confirmer l'aspect non intrusif de notre solution vis-à-vis de la mise en œuvre des applications.

8.3.1.3 Projection des composites

Tout comme le concept de connecteur, le concept de composite n'existe pas dans le CCM. La projection que nous proposons ici est donc un choix de mise en œuvre. Toutefois, cette mise en œuvre respecte les règles du CCM. La définition d'un composite relève donc plus du niveau méthodologie d'utilisation du CCM que du niveau modèle et technique. Notre volonté est toujours d'utiliser les technologies « *as it* » pour ne pas avoir à définir des extensions non standards (et donc sortir du cadre strict de ces technologies), ce qui réduirait l'utilisabilité et la portabilité de notre solution. Ainsi, tout développeur d'applications à base de composants CCM peut développer des applications dans le cadre de CODeX.

Actuellement, la projection d'un composite vers le CCM ne comprend que la projection de ses ports. Seule la réification du composite au sein du référentiel existe, il n'y a pas d'instance au niveau applicatif. La projection de mise en œuvre est donc uniquement réalisée si le composite contient des ports. Dans ce cas, pour chaque port du composite, un type de composant CCM est défini. Ce type de composant n'offre qu'un port, que ce dernier soit d'entrée ou de sortie, qui représente les interactions entre le composite et l'extérieur. La partie « interne », c'est-à-dire la connectivité entre ce port et les composants constituant le

composite, n'est pas automatiquement générée. Cette dernière doit être écrite manuellement. Ceci découle de la définition de notre méta-modèle et de l'absence de composites dans le CCM.

La nécessité de mettre en œuvre les ports de composites repose sur deux constatations. Premièrement, ces ports offrent une vision simplifiée du composite sans en montrer la structure, c'est-à-dire le patron de conception façade [29]. Il est donc nécessaire de pouvoir définir des traitements au sein des ports. Une invocation sur un port de composite va donc potentiellement être réalisée par plusieurs invocations sur les ports d'instances de composants constituant le composite. Deuxièmement, pour des raisons de définition du méta-modèle et pour respecter la propriété d'encapsulation, il ne doit pas y avoir de connexion directe entre deux instances de composants faisant partie de deux composites distincts. Toutes les connexions inter-composites doivent ainsi passer par des ports de composites.

```
component AProxy {
  provides A ref ;
} ;
component BProxy {
  uses B ref ;
} ;
component CProxy {
  uses C ref ;
} ;
```

FIG. 8.9 – *Projection des ports d'un composite en OMG IDL 3*

Dans le cas d'un composite constitué de plusieurs instances de composants et défini comme offrant un port d'entrée de type A et deux ports de sortie de type B et C, tous les trois en mode synchrone, la projection en OMG IDL 3 sera telle que définie dans la figure 8.9. La définition pour le port d'entrée A devra être complétée par un ensemble de réceptacles servant à connecter cette mise en œuvre de port avec les composants internes au composite. Si le port A utilise deux instances de composants au travers de leurs ports D et E, alors la définition du type de composant `AProxy` est enrichie de deux réceptacles de types D et E, cette dernière manipulation étant réalisée manuellement.

8.3.2 Production des interfaces OMG IDL 3

Contrairement aux autres générateurs, la génération d'interfaces OMG IDL 3 ne s'appuirait par sur une version XMI des architectures, mais sur la représentation des architectures contenues dans le référentiel. La production des interfaces OMG IDL 3 serait réalisée à partir d'un outil externe au référentiel. Cet outil parcourerait la définition d'une architecture, et pour chaque élément produirait l'interface OMG IDL 3 associée en suivant les règles de projection que nous venons de décrire. Ce parcours serait récursif et débuterait sur le composite de plus haut niveau, celui qui définit l'application. Ensuite, le générateur parcourerait tous les composites, composants et connecteurs contenus dans ce composite et ainsi de suite. Les interfaces ainsi produites seraient ensuite compilées à l'aide des outils d'OpenCCM, ou tout autre plate-forme, pour générer les souches et squelettes de l'application afin de les mettre en

œuvre.

8.4 Dynamique des applications

8.4.1 Mise en œuvre des actions architecturales

Cette section présente dans un premier temps la mise en œuvre des opérations architecturales de base. Ces opérations représentent les traitements élémentaires qui, composés, permettent de définir les actions architecturales.

8.4.1.1 Opérations architecturales de base

Contrairement aux concepts *composite*, *composant*, *port* et *connecteur*, les opérations architecturales de base ne sont pas spécialisées par héritage. En effet, ces instances « restent » au niveau M1. Ce sont elles qui vont, à partir de la définition de l'architecture, agir (création et destruction) sur les instances applicatives et sur leurs représentants. Le rôle de ces opérations est à la fois de gérer les instances applicatives et leur représentation. Les opérations architecturales de base sont toutes pourvues d'une méthode `eval ()` définie dans le méta-modèle. C'est la mise en œuvre de cette opération que nous allons discuter ici. Nous ne présenterons que les opérations de création d'instance de composant et d'établissement de connecteurs. Dans le cas des connecteurs, nous ne présentons ici que la mise en œuvre des connecteurs synchrones. C'est exclusivement ce type de connecteurs que nous avons manipulé.

Opération de création d'une instance de composant L'opération de création d'une instance de composant représente le déploiement de cette instance dans le système. La mise en œuvre de ce déploiement se décompose en deux parties. La configuration du site d'exécution, c'est-à-dire la création du conteneur et de la maison de composant, est effectuée par le représentant du site d'exécution, *i.e.* l'instance de `HostDefCCM`. La création à proprement parler de l'instance de composant CCM est réalisée directement par l'opération `eval ()` de la classe `OpCreate`. Sa mise en œuvre est illustrée dans la figure 8.10.

L'évaluation de l'opération de création d'une instance de composant commence par créer le représentant au sein du référentiel de cette instance, puis ajoute ce représentant au sein du représentant du composite. Pour cela, elle utilise la définition de cette instance de composant (de niveau M1) afin de connaître son nom et ses ports, pour en créer leur représentation au niveau M0. Ensuite, le site d'exécution de cet instance est retrouvé grâce à l'association `RunsOn` et à sa représentation dans le référentiel. De la même manière, l'archive contenant l'implémentation de composant est déterminée *via* l'association `IsImplementedBy`. Une fois ces deux éléments connus, le représentant du site d'exécution est utilisé pour, à la fois, charger l'archive localement, et configurer l'environnement d'exécution (c'est-à-dire création d'un conteneur et d'une maison de composant). Enfin, la référence de la maison est exploitée pour demander la création de l'instance de composant CCM au niveau applicatif. Cette référence étant ajoutée à la connaissance de sa représentation pour des manipulations futures.

Opération d'établissement d'un connecteur La mise en œuvre d'un connecteur synchrone et n'incluant pas de traitement est relativement simple. Une instance de niveau M0

```
proc eval (self) {
  # création de la représentation de l'instance de composant
  def  = self.pkg.acts_upon_component_ref.component (self) ;
  inst = myadlccm.component_def_ccm_ref.create (def) ;

  # ajout de la représentation dans le composite
  self.compositeccm.add_component_ref (inst) ;

  # récupération du site d'exécution et de l'archive a utiliser
  host = self.pkg.runs_on_ref.host (inst) .host ;
  arch = self.pkg.is_implemented_by_ref (inst) .archive ;
  # création de la maison de composant
  home = host.configure (arch.location (), arch.bootstrap ()) ;

  # création de l'instance applicative de composant
  inst.ref = home.create_component () ;
}
```

FIG. 8.10 – *Mise en œuvre de l'opération* `create` *pour le CCM (extrait)*

représentant un connecteur exploite la connaissance dont elle dispose sur les deux ports à interconnecter. Le port connu au travers de l'association MOF *from* décrit le réceptacle de l'une des instances de composant CCM. Le port connu au travers de l'association *to* représente quant à lui la facette de l'autre instance de composant CCM. Connaissant la représentation de ces deux ports, il est possible d'agir sur les instances de composants CCM pour récupérer la référence de la facette et la connecter au réceptacle. La figure 8.11 illustre cette mise en œuvre avec le langage IDLscript.

Comme pour la création des instances de composants, la première étape est de créer la représentation du connecteur au niveau M0. Pour cela, on utilise la définition du connecteur (au niveau M1) afin d'en créer une version (au niveau M0) représentant le lien effectif entre les deux instances de composants CCM. Cette représentation est ensuite ajoutée à la représentation du composite (au niveau M0). Puis, les deux ports à connecter sont recherchés dans cette représentation de composite afin d'établir les relations entre la représentation du connecteur et les représentations des deux ports. Il ne reste alors qu'a réaliser la connexion effective entre les deux instances de composants CCM, c'est-à-dire connecter la facette de l'une des instances au réceptacle de l'autre. La représentation d'un port de type réceptacle possède une référence non pas sur le réceptacle, mais sur l'instance de composant (référence de base). Ceci est dû au fait que les receptacles n'ont pas de référence dans le cadre du CCM. Le réceptacle est réifié par deux opérations (connexion et déconnexion) disponibles sur l'instance de composant CCM. Nous utilisons ici les opérations génériques `connect` pour établir la connexion avec le réceptacle.

8.4.1.2 Actions définies par l'architecte

Tout comme les opérations de base architecturales, les actions ne sont pas spécialiées par héritage. Les actions architecturales sont définies comme des séquences d'opérations de base

```
proc eval (self) {
  # création de la représentation du connecteur
  def = self.pkg.acts_upon_connector_ref.connector (self) ;
  cnx = myadlccm.connector_def_ccm_ref.create (def) ;

  # ajout de la représentation du connecteur dans le composite
  self.compositeccm.add_connector_ref (cnx) ;

  # recherche du port "from" de l'instance de niveau M0 puis
  # ajout dans la représentation de la connecteur
  from = ... ;
  myadlccm.from_ref.add ([cnx, from]) ;
  # recherche du port "to" de l'instance de niveau M0 puis
  # ajout dans la représentation du connecteur
  to = ... ;
  myadlccm.to_ref.add ([cnx, to]) ;

  # établissement effectif de la connexion au niveau CCM
  to.ref.connect (to.name(), from.ref) ;
}
```

FIG. 8.11 – *Mise en œuvre de l'opération* `OpBind` *pour le CCM*

et donc définies au même niveau. Les actions peuvent être spécialisées par rapport à la version de base présentée dans la section 7.3.4, en modifiant directement leur mise en œuvre. Cette approche est similaire à la spécialisation des opérations de base. Toutefois, le comportement d'une action est abstraitement définissable et il ne devrait être que rarement nécessaire de les spécialiser.

Les actions architecturales peuvent être regroupées en deux catégories. En premier lieu, l'action de déploiement de l'architecture initiale. Les opérations de base s'appliquent alors sur les instances de composants et les connecteurs définissant la structure de l'architecture initiale pour déployer celles-ci. Ensuite, les actions de reconfiguration sont toutes celles qui modifient cette architecture initiale. Dans ce cas, les opérations de base ajoutent de nouvelles méta-instances à la définition de l'architecture et déploient les instances de composants CCM correspondantes et leurs interconnexions.

8.4.2 Discussion sur la mise en œuvre du déploiement

8.4.2.1 Mise en œuvre du processus de déploiement du CCM

Le processus de déploiement défini dans la spécification du CCM est basique. Les applications sont définies comme des assemblages de composants. Ces assemblages sont décrits au sein de fichier OSD (*Open Software Descriptor*, un vocabulaire XML) et exploités par un outil de déploiement. Cet outil va, de manière séquentielle en respectant l'ordre de déclaration du fichier, installer les archives d'implémentation sur leurs sites d'exécution, configurer les serveurs de composants, créer les instances de composants, établir les connexions entre instances, configurer les instances et faire basculer l'application en phase d'exploitation (c'est-à-dire si-

gnaler à toutes les instances que leur configuration est terminée).

Le déroulement du processus de déploiement est défini à l'aide d'une action architecturale. Cette action définit un ordre de déploiement pour les instances de composants et pour l'établissement des connexions au sein d'un composite. Cet ordre est implicitement défini par la séquence d'opérations de base représentant la création des instances de composants et l'établissement des connexions entre ces instances. Cette action est définie par le déployeur d'applications. L'ordre de déploiement des instances de composants n'est pas stratégique à ce niveau. Ce qui est important, c'est l'ordre dans lequel les instances de composants passent de leur phase de configuration à leur phase d'exécution. Dans le cadre du CCM, cette transition est explicite par l'utilisation de l'opération `configuration_complete` sur les instances de composants. Une fois que toutes les opérations de base de l'action architecturale `deploy ()` sont évaluées, les instances de composants CCM sont configurées en respectant l'approche du CCM.

8.4.2.2 Utilisation d'une fonction de courtage

Dans ce que nous venons de voir, le choix des archives à utiliser et des sites d'exécution est fixé. Cette approche ne répond pas à tous les besoins. Dans certains contextes, le déploiement ne peut être anticipé et requiert une évaluation dynamique, dépendant du contexte dans lequel il est réalisé. Le projet CESURE [1] [54, 82, 84] est une mise en œuvre de cette approche. Une même application était déployée dans des environnements hétérogènes et pouvant changer radicalement d'un déploiement à l'autre : une fois sur un PDA de type iPaq, la fois suivante sur une station de travail. Dans ce contexte, il n'est pas possible de fixer au sein de la description de l'architecture le site d'exécution ou l'archive pour toutes les instances de composants.

Il est possible, en mettant en œuvre une fonction de courtage, de laisser un, voire deux degrés de liberté. On fixe, par exemple, les sites d'exécution et on recherche au moment du déploiement quelle archive, contenant l'implémentation d'une interface de composant donnée, est la plus appropriée. Pour un site donné on évalue à l'exécution s'il est capable d'exécuter des implémentations en Java ou en C++. La seconde possibilité est de fixer l'archive de composant à utiliser et de choisir au moment du déploiement le site d'exécution le plus adapté pour supporter cette implémentation. Indépendamment du projet CESURE, un autre intérêt de cette approche est, par exemple, le cas d'une application nécessitant une certaine qualité de service. Les sites les plus aptes à garantir la qualité de service requise sont recherchés au moment du déploiement de l'application.

Dans le cas de l'utilisation d'une fonction de courtage, les ressources, dans le contexte courant les sites d'exécution et archives de composants, doivent être enregistrées auprès de cette fonction. Les ressources sont décrites à l'aide de contrats de courtage [51, 52, 43] qui vont permettre de les rechercher au moment du déploiement. Ainsi, une recherche dynamique, en fonction des besoins, est mise en œuvre. Ceci permet de déployer une même application dans des contextes totalement différents. Enfin, avec une fonction de courtage intelligente, il est possible de définir deux degrés de liberté. Dans ce cas, la fonction de courtage doit faire une synthèse des implémentations et des sites d'exécution disponibles pour fournir des

1. Le projet RNRT CESURE, commencé en novembre 1999 et terminé en novembre 2001, regroupait les laboratoires suivants : Gemplus Research Labs, projet SIRAC de l'INRIA Rhones-alpes, équipe réseau de l'INT Evry, équipe GOAL du LIFL.

couples site/archive qui correspondent, pour que l'implémentation puisse être utilisée par le site d'exécution. Si cette synthèse ne peut être effectuée par la fonction de courtage, elle doit être mise en œuvre dans la procédure de déploiement.

8.5 Utilisation de l'environnement

Les interactions avec le référentiel d'architectures ont été réalisées à l'aide du langage OMG IDLscript. Ce langage permet de définir simplement des architectures et de les manipuler à l'aide d'une console textuelle. Cette section présente comment, à l'aide de ce langage, l'architecture du dîner des philosophes est définie dans le référentiel en respectant la séparation des préoccupations définie au niveau du méta-modèle.

1. L'*architecte* définit la structure de l'application. Il définit les différents composants de l'architecture (trois philosophes et trois fourchettes) ainsi que leur interconnexion.
2. L'architecture est exploitée par le *développeur* pour produire les interfaces OMG IDL3 associées et réaliser la mise en œuvre des composants.
3. L'*intégrateur* précise les implantations de composants, réalisées par le développeur, à utiliser pour déployer l'application.
4. Le *placeur* de composant précise sur quel site d'exécution les instances de composants doivent être déployées.
5. Le *déployeur* de composants décrit le processus de déploiement en utilisant une action architecturale.

8.5.1 Définition de l'architecture

Nous illustrons dans cette section comment une architecture peut être définie de manière incrémentale à l'aide de notre méta-modèle. La définition de l'application « dîner des philosophes » revient à définir un *composite* représentant le dîner. Au sein de ce *composite*, un ensemble de composants et de connecteurs sont définis. La figure 8.12 illustre la définition du *composite* `diner` et d'un *primitif* `philosophe1` en OMG IDLscript. Une fois défini, `philosophe1` est ajouté aux *primitifs* du composite.

```
base = rep.create_architectural_base_package () ;
diner = base.composite_def_ref.create_composite_def () ;
diner.name = "diner" ;

phi1 = rep.primitive_def_ref.create_primitive_def () ;
phi1.name = "philosophe1" ;
diner.add_primitive_ref (phi1) ;
```

FIG. 8.12 – *Définition du composite* `diner` *et d'un primitif* `philosophe1`

La figure 8.13 présente la définition d'un *port* représentant une main de philosophe et son ajout dans la liste des ports du `philosophe1`. Le *port* `main_gauche` représente la main gauche du philosophe qui sera connecté à une fourchette. Ce *port* contient deux opérations :

`prendre` et `poser`. Ces deux opérations ne prennent pas d'argument et n'ont pas de valeur de retour. La définition de la main droite du philosophe n'est pas présentée ici car elle est exactement identique à la définition de la main gauche. Pour ne pas avoir à refaire plusieurs fois les mêmes définitions, l'architecte dispose d'une opération `clone()` qui fait une copie conforme d'une définition contenue dans le référentiel ; que ce soit un *composite*, un *primitif* ou un *port*. Cette opération n'est pas détaillée ici ; elle repose sur un parcours récursif d'arbre réifiant une définition. Une fois cloné, le nom du nouveau *port* est affecté : `main_droite`.

```
maing = base.output_port_def_ref.create_output_port_def () ;
maing.name = "main_gauche" ;
phi1.add_port_ref (maing) ;

op1 = base.operation_def_ref.create_operation_def () ;
op1.name = "prendre" ;
op1.type = "void" ;
maing.add_operation_ref (op1) ;

op2 = base.operation_def_ref.create_operation_def () ;
op2.name = "poser" ;
op2.type = "void" ;
maing.add_operation_ref (op2) ;

maind = clone (maing) ;
maind.name =  "main_droite" ;
phi1.add_port_ref (maind) ;
```

FIG. 8.13 – *Définition du port* `main_gauche` *du composant* `philosophe1`

La définition des autres *primitifs* philosophes de l'architecture sont crées par clonage du `philosophe1` et affectation de leurs noms. La définition des *primitifs* fourchettes sont semblables à la définition des philosophes et ne sont donc pas détaillés ici. La figure 8.14 présente la définition d'un connecteur entre le port `main_gauche` du *primitif* `philosophe1` et le port `four1_p` d'un *primitif* fourchette. L'objet `cnx1`, définissant le connecteur, est créé et ajouté au composite. Puis les deux relations entre le connecteur et les ports sont fixés.

```
cnx1 = base.connector_def_ref.create_connector_def () ;
diner.add_connector_ref (cnx1) ;

cnx1.set_to_ref (four1_p) ;
cnx1.set_from_ref (maing) ;
```

FIG. 8.14 – *Définition d'un* connecteur *entre deux* primitifs

La définition du *composite* `diner`, des *primitifs* `philosophe*` et `fourchette*` et des *connecteurs* représente la définition de l'architecture de base de l'application. Cette définition va maintenant être exploitée et enrichie par les autres acteurs du processus logiciel.

8.5.2 Définition des interfaces de mise en œuvre

L'application du « diner des philosophes » utilise deux types de composants primitifs : les philosophes et les fourchettes. Tous les primitifs étant définis par clonage et non étendus après clonage, l'architecte considère que leurs types sont similaires. Ainsi, il précise donc au(x) développeur(s) de réaliser une implantation pour chacun de ces types. Le choix d'avoir un même type pour tous les philosophes est relativement dirigé par leur définition. Le choix d'utiliser une seule mise en œuvre est arbitraire. On pourrait souhaiter avoir des comportements différents pour chaque philosophe. (Cette remarque s'applique aussi aux fourchettes.) Le « contrat » entre l'architecte et le(s) développeur(s) correspond aux interfaces OMG IDL 3 de ces deux types. La projection de la définition de ces primitifs est illustrée dans la figure 8.15. Cette projection a été réalisée manuellement.

```
interface IFourchette {
  void prendre () ;
  void poser () ;
} ;

component Philosophe {
  uses IFourchette main_gauche ;
  uses IFourchette main_droite ;

  attribute string name ;
} ;

component Fourchette {
  provides IFourchette manche ;
} ;
```

FIG. 8.15 – *Projection en OMG IDL 3 des interfaces de* primitifs

Un développeur va utiliser ce contrat OMG IDL 3 et un environnement de développement pour le modèle de composants CORBA [2]. La compilation de cette interface va produire les squelettes de mise en œuvre des *primitifs* `Philosophe` et `Fourchette` et les souches d'utilisation. Une fois les squelettes générés, il ne reste plus au développeur qu'à réaliser l'implantation fonctionnelles ce ceux-ci ; c'est-à-dire à écrire les traitements qui seront réalisés par une instance de composant.

8.5.3 Spécification du placement et des archives

Pour chaque *primitif* de l'application, l'*intégrateur* associe l'archive de composant contenant son implantation. La figure 8.16 présente la mise en relation du *primitif* `philosophe1` avec l'implantation réalisée par le développeur. Cette mise en relation sera similaire pour tous les *primitifs* du *composite* `diner`. Dans un premier temps, la représentation de l'archive contenant l'implantation est définie par rapport au package de base d'implantation. Les représentations de l'interface et de l'implantation contenues dans cette archive sont définies

2. Par exemple, notre plate-forme OpenCCM. . . :)

et mises en relation avec celle-ci. Enfin, l'association entre l'archive et le *primitif* est créée et configurée, à l'aide du package d'annotation d'implantation. Pour cette dernière opération, la recherche de la définition du package architectural de base `base` et du *primitif* `philosophe1` dans le référentiel ne sont pas présentées (parcours de la liste des packages contenus dans le référentiel et de la liste des *primitifs* contenus dans le *composite* `diner`). Cette définition d'archive pourra être utilisée pour tous les *primitifs* philosophes ; ils sont implantés de la même manière. La dernière ligne de ce script sera donc utilisée avec les autres *primitifs* en remplaçant `phi1` par l'instance définissant ceux-ci.

```
# creation du plan d'implantation
archives = rep.create_implementation_base_pkg () ;

# définition d'une archive de composant
arch = archives.archive_def_ref.create_archive_def () ;
arch.url = "http://www.lifl.fr/~marvie/arch/philosophe.zip" ;

# definition de la représentation d'une interface
itf = archives.itf_def_ref.create_itf_def () ;
itf.name = "Philosophe" ;
itf.file = "philosophe.idl" ;

# association entre archive / interface
archives.is_described_by_ref.add (arch, itf) ;

# définition d'une implantation de composant
impl = archives.impl_def_ref.create_impl_def () ;
impl.lang    = "java" ;
impl.runtime = "orbacus-4.1.0" ;
impl.version = "jdk-1.3.1" ;
impl.file    = "philosophe.jar" ;

# association entre archive / implantation
archives.is_implemented_by_ref.add (arch, impl) ;

# a_impl = recherche du package d'annotation d'implantations
# phi1 = recherche du primitif dans le composite diner
# association entre philosophe1 / archive de mise en {\oe}uvre
a_impl.is_implemented_by_ref (phi1, arch) ;
```

FIG. 8.16 – *Mise en relation du primitif* `philosophe1` *avec son implantation*

La mise en relation d'un *primitif* avec son site d'exécution est similaire à la mise en relation d'un *primitif* avec son implantation. Le *placeur* va définir la représentation d'un site d'exécution et associer le(s) serveur(s) de composants de ce site au(x) *primitif(s)* concerné(s). Ces opérations sont illustrées dans la figure 8.17. La définition d'un site d'exécution est réalisée à l'aide du package de base du placement. Elle est ensuite configurée pour préciser les caractéristiques de ce site. La représentation du serveur de composant disponible sur le site `alfri` est définie, de manière similaire, et mise en relation. Puis, la connexion réseau du site

alfri est précisée. Enfin, le serveur de composants alfri-srv1 et associé avec le primitif phi1 pour préciser son site d'exécution. La définition du serveur de composant alfri-srv1 peut être réutilisée pour tous les *primitifs* qui s'exécutent sur ce site. La définition de la connexion réseau (net) peut être réutilisée par tous les sites d'exécution connectés au même réseau.

```
# création du plan de placement
location = rep.create_location_base_pkg () ;

# definition d'un site d'exécution
host = location.host_def_ref.create_host_def () ;
host.id      = "alfri.lifl.fr" ;
host.type    = "workstation" ;
host.version = "linux-2.4.18" ;

# definition d'un serveur de composants
srv = location.server_def_ref.create_server_def () ;
srv.id      = "alfri-srv1" ;
srv.lang    = "java" ;
srv.version = "1.3.1" ;

# création de l'association host / server
location.supports_ref.add (host, srv) ;

# definition d'une connexion réseau
net = location.network_def_ref.create_network_def () ;
net.kind      = "802.3" ;
net.bandwidth = "10 Mb" ;

# création de l'association host / réseau
location.is_connected_to_ref.add (host, net) ;

# a_loc = recherche du package d'annotation de placement
# phi1 = recherche du primitif dans le composite 'diner'
# création de l'association philosophe1 / serveur
a_loc.runs_on_ref (phi1, srv) ;
```

FIG. 8.17 – *Mise en relation du primitif* ***philosophe1*** *avec son serveur d'exécution*

8.5.4 Définition de l'action de déploiement

L'ensemble de l'architecture du « dîner des philosophes » est maintenant réalisée. Il ne reste plus qu'à définir son processus de déploiement afin de réaliser des instances de cette application. Le *déployeur* va définir une action architecturale sur cette version de l'architecture pour spécifier ce processus. La figure 8.18 présente un extrait de la définition de l'action architecturale deploy sur le composite diner. Cet extrait précise la création des *primitifs* philosophe1, fourchette1 et fourchette2 ; ainsi que des connecteurs. La création des autres *primitifs* et l'établissement des autres connexions et similaire.

La définition de l'action architecturale `deploy` commence par la création et la configuration de l'objet la représentant ; à l'aide du package de base de la dynamique. Ensuite, les opérations élémentaires de cette action sont définies. L'opération élémentaire de création du *primitif* `philosophe1` se décompose en deux étapes. La définition de l'opération de création du *primitif* `fourchette1` (`op2`) est identique.

- Premièrement, la définition d'un objet représentant une opération de création d'un *primitif* (`op1`) et son ajout à la liste des opérations de l'action architecturale.
- Deuxièmement, la mise en relation de cette opération de création avec la représentation du primitif `philosophe1` dans le référentiel. Cette mise en relation se fait par l'ajout d'une entrée pour l'association *ActsUponCmp* dans le package architectural de base.

La définition d'une opération d'établissement de connexion est similaire à la définition d'une opération de création d'un *primitif*. La différence tient dans la création d'un objet de type *Bind* plutôt que *Create* ; et dans la mise en relation de l'opération avec une définition de connecteur plutôt qu'avec une définition de *primitif*. Enfin, la dernière ligne de cet extrait de script spécifie l'ajout de l'opération `deploy` au composite `diner`.

```
# a_dyn = recherche du package d'annotation de la dynamique
# diner = recherche du composite 'diner' dans le référentiel
dynamism = rep.create_dynamism_base_pkg () ;
deploy = dynamism.action_def_ref.create_action_def () ;
deploy.name = "deploy" ;

op1 = dynamism.create_ref.create_create () ;
dynamism.is_defined_as_ref.add (deploy, op1) ;
# phi1 = recherche de 'philosophe1' dans le composite 'diner'
a_dyn.acts_upon_cmp_ref.add (op1, phi1) ;

op2 = dynamism.create_ref.create_create () ;
dynamism.is_defined_as_ref.add (deploy, op2) ;
# fou1 = recherche de 'fourchette1' dans le composite 'diner'
a_dyn.acts_upon_cmp_ref.add (op2, fou1) ;

op3 = dynamism.bind_ref.create_bind () ;
dynamism.is_defined_as_ref.add (deploy, op3) ;
# cnx1 = recherche du connecteur entre 'philosophe1' et
# 'fourchette1' dans le composite 'diner'
a_dyn.acts_upon_cnx_ref.add (op3, cnx1) ;

# etc. pour les autres primitifs et connecteurs

# ajout de l'action architecturale au composite 'diner'
a_dyn.offers_ref (diner, deploy) ;
```

FIG. 8.18 – *Définition de l'opération* `deploy` *sur le composite* `diner`

L'évaluation du processus de déploiement de l'application « dîner des philosophes » peut se résumer (quasiment) en une seule ligne (voir à la fin de ce paragraphe). Une fois ce processus évalué, il ne reste plus qu'à « basculer » les instances applicatives de composants de la phase de

configuration vers la phase d'exploitation, en invoquant sur chacune d'entre elles l'opération `configuration_complete` définie dans le modèle de composants CORBA.

```
# deploy = recherche de l'action 'deploy' dans le composite diner
deploy.eval () ;
```

8.6 Conclusion

Après avoir vu dans les chapitres précédents comment définir le méta-modèle d'un ADL et produire son environnement associé, nous venons de discuter **comment ce référentiel est spécialisé pour une utilisation dans le contexte d'une technologie particulière**, ici le *CORBA Component Model*. La spécialisation du référentiel abstrait est donc essentiellement la définition du « lien » entre la définition des architectures et les instances d'applications. Ce lien est réalisé par le niveau M0 de notre pile de méta-modélisation, c'est-à-dire les représentations dans le référentiel des instances applicatives.

En addition de cette spécialisation du référentiel, nous avons discuté de **la projection des architectures d'applications vers les interfaces des composants** utilisées en OMG IDL 3. Cette projection utilisée dans le cadre du CCM fournit une base pour la mise en œuvre des applications. Elle est donc destinée à être utilisée par les développeurs et les intégrateurs pour fournir les archives de composants. La spécialisation pour le CCM est uniquement un exemple de spécialisation de notre méta-modèle. En effet, ce même ADL pourrait être projeté vers une autre technologie. Cette projection a été expérimentée manuellement et pourrait être automatisée.

L'adaptation d'un méta-modèle d'ADL pour des modèles de composants, sans modification de ceux-ci, est possible bien que parfois non totalement automatisable. C'est le cas dans notre contexte de la projection des ports de composites. L'intérêt de ne pas étendre le modèle technologique est multiple : pas de remise en cause du modèle, pas de nouveau développement pour supporter des extensions propriétaires, directement utilisable par des spécialistes du modèle, outils / mise en œuvre externes à CODeX utilisable. De plus, il est ainsi possible de tirer bénéfice de toute optimisation déjà réalisée dans la mise en œuvre du modèle technologique.

Le point le plus sensible de la spécialisation est la mise en œuvre des opérations de base architecturales. En effet, elle n'est pas automatisable et doit être **réalisée par des spécialistes du modèle technologique visé**. Toute la mise en œuvre du déploiement repose sur ces opérations de base. L'intérêt de leur utilisation est d'offrir la simplicité de définition des actions architecturales par composition. Ainsi, toute la complexité des traitements est masquée par ces opérations.

Nous avons enfin pu voir dans ce chapitre que, malgré l'absence d'une interface graphique dans notre prototype, la définition et la manipulation d'architectures est réalisable avec un environnement défini à l'aide de CODeX associé au langage OMG IDLscript. Ces extraits de scripts ont montré que :

- l'architecture de base est partagée par tous les acteurs (utilisation du package `base`) ;
- la séparation des préoccupations est effective, chaque acteur utilisant une interface de package dédiée à son activité.

Les extraits de scripts présentés ici peuvent de manière quasi-directe être utilisés au travers de l'environnement graphique CorbaWeb [58, 59]. Au-delà de cette mise en œuvre directe, ces scripts représentent les actions qui doivent être associées, dans un outil graphique, aux boutons de définition et de manipulation des éléments architecturaux. Le gain essentiel apporté par un outil graphique est de transformer la représentation mentale reposant sur les affichages textes en une « nifty » représentation visuelle des architectures. Représentation qui simplifie d'autant plus la manipulation des architectures par les acteurs d'un processus logiciel. L'ensemble de ces expérimentations illustre les mécanismes à mettre en œuvre pour manipuler des architectures, et montre que ces mécanismes sont simples grâce à la séparation des préoccupations dans la réification des architectures.

Troisième partie

Conclusion et perspectives

Chapitre 9

Conclusion et perspectives

Les domaines d'activité exploitant des systèmes d'information ou automatisés sont nombreux et variés. Le commerce électronique, les applications de télécommunication, le trafic du contrôle aérien, ou l'informatique embarquée dans les véhicules ne sont que quelques exemples parmi tant d'autres. Chaque domaine regroupe un certain nombre de préoccupations auxquelles les applications doivent répondre. Parallèlement, les contraintes de conception et de production de ces applications sont variables et requièrent des outils spécifiques. Chaque domaine dispose d'une vision particulière du processus logiciel d'ingénierie des applications.

Les modèles de composants et les langages de description d'architectures représentent des approches qui tendent à se généraliser pour concevoir et mettre en œuvre des applications dans une grande majorité de domaines applicatifs. Nous avons discuté dans la première partie de ce document des bénéfices et limitations de ces approches. Nous avons mis en avant le fait que ces propositions technologiques étaient soit trop spécifiques pour être utilisées dans plusieurs domaines, soit trop génériques pour bien répondre aux spécificités de chaque domaine applicatif. Enfin, nous avons souligné les possibilités réduites d'adaptation de ces propositions à un domaine applicatif particulier.

La motivation de notre travail a été de fournir une réponse à ces limitations en proposant un cadre de travail pour définir et produire un ensemble de moyens supportant la conception et l'exploitation d'architectures logicielles. Ces moyens ont pour objectif de répondre de façon spécifique aux préoccupations d'un processus logiciel afin de répondre au mieux aux besoins de ses acteurs. Pour cela, nous avons proposé d'exploiter les techniques de méta-modélisation et de mettre en œuvre la séparation des préoccupations pour proposer un support à la manipulation d'architectures logicielles à base de composants.

Dans un premier temps, nous résumons les propositions discutées dans ce document ainsi que leurs bénéfices. Nous discutons ensuite des perspectives qui ont émergées de nos travaux et expérimentations.

9.1 Travaux réalisés

La contribution principale de cette thèse est la proposition d'un cadre de travail pour fournir des environnements de manipulation d'architectures adaptées aux particularités des processus logiciels. Ce cadre de travail regroupe une méthodologie de définition et un ensemble de prototypes d'outils destinés à produire de façon automatisée ces environnements. Ces deux constituants forment l'environnement CODeX.

9.1.1 Méthodologie

La première contribution de ce travail est la proposition d'une méthodologie de conception d'un environnement destiné à structurer la manipulation d'architectures logicielles par les acteurs d'un processus logiciel. Cette structuration vise à faciliter l'activité et la collaboration de ces acteurs. Pour atteindre cet objectif, nous avons à la fois mis en œuvre la séparation des préoccupations et utilisé les techniques de méta-modélisation.

Les technologies de mise en œuvre des architectures sont nombreuses, et leur évolution est rapide. De plus, le meilleur est toujours « à venir », et il est bon de pouvoir en profiter lorsqu'il apparaît. Dans le but de pérériser les architectures définies à l'aide de ces environnements, nous proposons de capitaliser leur description dans une forme indépendante des technologies de mise en œuvre.

- Les techniques de méta-modélisation permettent de définir les concepts relatifs à un domaine d'activité, ainsi que leurs relations. Nous avons utilisé ces techniques pour spécifier les concepts des environnements de manipulation d'architectures. Ainsi, nous proposons une approche qui tend à spécifier l'environnement minimal et en adéquation avec les besoins architecturaux d'un domaine d'activité. L'environnement n'impose plus au processus logiciel un ensemble de préoccupations prises en compte, mais il offre un support aux préoccupations identifiées dans ce processus. Ce point respecte la maxime[1] « C'est à l'outil informatique de s'adapter à l'usager et non l'inverse ». **La proposition CODeX montre que les technologies de méta-modélisation représentent un bon support à la définition de moyens de définition d'architectures adaptés aux préoccupations des processus logiciels.**
- La séparation des préoccupations est une approche connue depuis de nombreuses années pour structurer et simplifier la conception et la production de logiciels. Toutefois, nombre de moyens de définition d'architectures logicielles, comme les langages de description d'architectures, ne suivent pas cette approche. Notre proposition CODeX prend en compte la séparation des préoccupations dans la définition des environnements. Cette séparation permet de répondre au problème de la collaboration des différents acteurs d'un processus logiciel. Si un certain nombre de préoccupations sont suffisamment indépendantes, il est alors possible pour chaque acteur, dans un cadre bien défini, de travailler de manière autonome. Ce cadre garantit que l'intégration des différentes préoccupations se fera sans accroc pour atteindre l'objectif global, ici l'architecture complète d'une application. Chaque acteur dispose ainsi de sa propre vision de l'architecture, ce qui simplifie son activité. **La proposition CODeX montre que la mise en œuvre de la séparation des préoccupations contribue à la fourniture de moyens structurés de définition d'architectures logicielles.**
- L'utilisation d'une forme réifiée des architectures permet de faciliter la manipulation de celles-ci. Il est possible d'interagir simplement et dynamiquement avec la définition d'une architecture. Les informations relatives à l'architecture sont disponibles dès leur définition et aisément partageables entre les différents acteurs d'un processus logiciel. L'utilisation d'une forme réifiée matérialise la constatation que l'architecture d'une application est centrale dans un processus logiciel. L'ensemble des activités des différents

1. Un peu ambitieuse il est vrai...

acteurs gravite autour de cette représentation. Cette approche introduit un second bénéfice en rendant disponible l'architecture d'une application au cours de son exécution. Elle peut servir de support aux activités telles que l'administration et la supervision. **La proposition CODeX montre que l'utilisation d'architectures dans leur forme réifiée contribue à la collaboration des acteurs d'un processus logiciel.**

– La séparation des préoccupations étudiées dans le cadre de ce travail structure principalement les préoccupations d'ordres architecturales. Un second type de séparation a aussi été mis en œuvre. La distinction entre préoccupations fonctionnelles et technologiques. En appliquant l'approche encouragée par la *Model Driven Architecture*, les architectures sont dans un premier temps exprimées indépendamment des technologies de mise en œuvre, puis exploitées dans le cadre d'une technologie particulière. Cette approche, destinée dans un premier temps à la modélisation d'applications, est aussi applicable pour la définition d'architectures logicielles. Elle permet de s'abstraire des contraintes technologiques et de rendre les architectures plus pérennes. La mise au second plan des considérations techniques permet aussi de rester en adéquation avec un processus logiciel. **La proposition CODeX propose une capitalisation des architectures logicielles indépendamment des technologies de mise en œuvre.**

9.1.2 Outillage

Afin d'expérimenter et de supporter cette méthodologie un certain nombre de moyens ont été définis ou prototypés.

– Le méta-méta-modèle de CODeX est une spécialisation du méta-méta-modèle MOF. Il définit un certain nombre de règles d'utilisation du MOF pour définir des méta-modèles d'ADLs. Ces règles servent de support à la structuration des ADLs en respectant la séparation des préoccupations architecturales. Ensuite, il définit la manière dont les différentes préoccupations sont intégrées pour fournir une version complète des ADLs. Enfin, le méta-méta-modèle de CODeX permet de définir des ADLs indépendamment de toute solution technologique de mise en œuvre. Ces ADLs sont ensuite spécialisés pour une technologie particulière de composants à l'aide de règles de projections. **Le méta-méta-modèle de CODeX représente le cadre de mise en œuvre de la méthodologie CODeX pour définir des méta-modèles structurés d'ADLs, indépendamment des technologies.**

– Les règles de projections présentées dans ce document montrent que la spécialisation d'un ADL pour une technologie particulière de composants logiciels est assez simple à mettre en œuvre. La projection pour le modèle de composants CORBA représente notre évaluation de cette spécialisation. L'utilisation d'une relation d'héritage représente la manière la plus directe de définition de cette spécialisation. Elle permet ensuite d'utiliser les outils de génération de référentiels pour produire un environnement de manipulation d'architectures spécialisé pour une solution technologique. **La définition de règles de projection permet de spécialiser simplement un ADL, qui plus est pour plusieurs modèles technologiques de composants.**

– La suite d'outils *CODeX M2 Tool* a été prototypée pour produire, à partir du méta-modèle d'un ADL, l'environnement de manipulation d'architectures associé. Ces outils

mettent en œuvre les projections définies dans la spécification du MOF. Ils montrent que les référentiels, définis à l'origine pour la manipulation de modèles, répondent bien aux besoins de manipulation des architectures. Ils montrent aussi que la spécialisation de ces référentiels est relativement simple à mettre en œuvre pour un usage particulier, qui sort un peu du cadre prévu initialement par le MOF. Les projections définies dans le MOF sont donc utilisables pour automatiser la production d'environnements d'exécution. Enfin, l'utilisation d'un langage interprété, IDLscript dans notre cas, permet de réaliser des référentiels flexibles et dynamiques. **Le prototype *CODeX M2 Tool* montre que les référentiels spécifiés par le MOF peuvent être utilisés pour la production automatisée d'environnements de manipulation d'architectures logicielles.**

9.2 Perspectives

Les perspectives à court de terme de nos travaux représentent un prolongement direct de ceux-ci.

- Une première perspective, relativement technique, de ces travaux est l'intégration dans notre plate-forme OpenCCM d'un environnement de définition et de manipulation des architectures produit à l'aide de CODeX. Cet environnement pourrait avoir comme double objectif de supporter la définition des applications à base de composants CORBA, ainsi que de supporter le déploiement et l'administration de celles-ci au travers d'actions architecturales. En addition de l'environnement de manipulation des architectures, le référentiel spécialisé, la fourniture d'une interface graphique simple est un besoin. Une fois expérimentée, la production de cet environnement peut être rendue automatisable. L'environnement CODeX pourra alors être enrichi de la génération d'outils graphiques à partir des méta-modèles pour disposer d'outils dédiés à un ADL. Enfin, pour que la boucle soit bouclée, la réalisation de référentiels à l'aide de composants CORBA (et non simplement d'objets) est une proposition qui émerge à l'OMG pour la spécification 2.0 du MOF. Ces perspectives pourront donc être utilisées pour expérimenter et contribuer à la définition de cette nouvelle approche.
- La définition d'un ADL avec l'approche actuelle de CODeX repose sur la définition de plans de base et d'annotation. Ces plan sont actuellement définis spécifiquement pour chaque ADL, et ne sont pas sujet à la réutilisation. Une seconde perspective de notre travail est la spécification d'un cadre de définition d'ADLs par composition de méta-modèles. Ce cadre représenterait une approche orientée composants pour la définition de méta-modèles. Un ensemble de méta-modèles spécifiant les préoccupations architecturales serait défini de manière générique, et configurable pour une utilisation particulière. A l'aide du modèle de composition, ces méta-modèles seraient composés pour définir l'ADL requis par un processus logiciel. Ceci rejoint l'approche de la programmation orientée aspects, la composition reviendrait à un tissage de méta-modèles. L'approche orientée composition de modèles permet d'accroitre la simplicité de définition d'un méta-modèle d'ADL, et d'améliorer le temps de mise à disposition d'un environnement de manipulation des architectures.
- Enfin, la séparation des préoccupations telle que nous l'avons expérimentée au niveau des méta-modèles n'est pas un besoin limité à notre problématique. La dernière perspective

de ce travail est d'envisager une généralisation de cette approche aux techniques de méta-modélisation en général. Cette généralisation pourrait se traduire sous la forme d'une méthodologie pour spécialiser un méta-méta-modèle, afin de fixer un cadre pour la définition de méta-modèles dans un domaine d'activité particulier. Un débouché de cette perspective pourrait, en plus de la méthodologie, contribuer au niveau de l'OMG autour de la MDA.

Les perspectives à plus long terme de ce travail partent du constat que les techniques de modélisation et de méta-modélisation tendent progressivement à se généraliser, aussi bien dans le monde académique que dans le monde industriel. Il est donc concevable de considérer aujourd'hui que les modèles seront demain présents un peu partout dans le domaine du génie logiciel et de l'informatique répartie. Bien que les techniques de (méta-)modélisation soient actuellement utilisées selon une approche « statique », le besoin de dynamicité va certainement apparaître progressivement. Le panorama informatique de demain sera donc certainement constitué d'un grand nombre de (méta-)modèles existants dans des référentiels répartis. La présence de ces (méta-)modèles entrainera certainement la présence d'un grand nombre de transformations entre ces modèles. Ces transformations existeront aussi bien au sein d'organisation, qu'entre organisations différentes. **Les transformations et les méta-modèles deviendront, au même titre que les modèles, des entités de première classe.** Cette vision soulève alors un certain nombre de questions :

- Comment doit être organisée la gestion, le contrôle ainsi que la maîtrise du nombre et de la diversité de ces (méta-)modèles et de ces transformations ?
- Comment peut-on associer des modes d'emploi à l'utilisation de ces transformations et à la composition des (méta-)modèles ?
- Nous avons commencé à expérimenter avec CODeX l'utilisation de référentiels contenant des traitements exécutables. L'ubiquité des (méta-)modèles et des transformations va certainement rendre cette préoccupation d'actualité.
  - En effet, comment faut-il organiser les (méta-)modèles et les transformations pour rendre ces dernières exécutables ?
  - Comment peut-on et doit-on sélectionner et utiliser une transformation ?
  - Une fois une transformation définie entre deux (méta-)modèles, son utilisation ne doit pas restée « statique ». A partir du moment où tout (méta-)modèle est réifié, comment peut-on rendre ces (méta-)modèles réactifs ? Comment peut-on, au travers de transformations, reporter directement les modifications apportées sur un (méta-)modèle à tous les (méta-)modèles résultant de l'application de ces transformations sur ce premier ?

Toutes ces questions nous ouvrent un large espace de recherche autour de l'ingénierie des (méta-)modèles et de leur exécutabilité.

Annexe A

Publications

Ce chapitre regroupe toutes les publications produites pendant cette thèse en tant qu'auteur principal ou associé, par ordre chronologique inverse. Nombre de ces publications sont disponibles à l'adresse : `http://www.lifl.fr/~marvie`.

Résumé des publications

Type	nombre
Chapitre de livre	1
Revue internationale	1
Revues nationales avec comité de sélection	3
Conférences internationales avec comité de sélection et actes (dont EDOC 2002, COOTS'01, DOA'00)	9
Atelier de travail international avec comité de sélection et actes	1
Activités internationales de standardisation	2
Conférences nationales avec comité de sélection et actes	2
Atelier de travail national avec comité de sélection et actes	1
Atelier de travail national sur invitation	1
Tutoriels internationaux et nationaux	2
Rapports de recherche et techniques	8
Total	**31**

Liste des publications

[**Edoc02**] Raphaël Marvie, Philippe Merle et Jean-Marc Geib, « *Separation of Concerns in Modeling Distributed Component-based Architectures* », Actes de la 6th IEEE International Enterprise Distributed Object Computing Conference (*EDOC 2002*), EPFL, Lausanne, Suisse, 17 - 20 septembre 2002. IEEE Press. ISBN : 0-7695-1742-0

[**Hermes02b**] Raphaël Marvie et Marie-Claude Pellegrini, « *Modèles de composants, un état de l'art* », publié dans « *Coopération dans les systèmes à objets* », numéro spécial de la revue *L'objet*, volume 8, num. 3, Paris, France, Septembre 2002. Editions Hermès.

[**Hermes02a**] Sylvain Leblanc, Raphaël Marvie, Philippe Merle et Jean-Marc Geib, « *TORBA : vers des contrats de courtage* », Chapitre du livre *Les intergiciels*, Avril 2002, Editions Hermès. ISBN : 2-7462-0432-0

[**Ike02**] Raphaël Marvie, Lev Kozakov et Yurdaer Doganata « *Towards Adaptive Knowledge Middleware* », Actes de l'International Conference on Information and Knowledge Engineering (*IKE'02*), Las Vegas, Nevada, Etats-Unis, 24 - 27 juin 2002.

[**Accord02**] Raphaël Marvie, Philippe Merle et Olivier Caron, « *Le modèle de composants CCM* », rapport technique du projet *RNTL ACCORD* Lot-1.1, avril 2002.

[**Lifl02**] Raphaël Marvie et Philippe Merle, « *CORBA Component Model: Discussion and Use with OpenCCM* », rapport technique LIFL, janvier 2002.

[**Cesure01**] Guy Bernard, Jean-Marc Geib, Daniel Hagimont, Vania Marangozova, Raphaël Marvie, Philippe Merle, Olivier Potonniée, Erik Putricz et Chantal Taconnet, « *CESURE - Rapport final* », rapport technique du projet *RNRT CESURE* 98-7, novembre 2001.

[**Jc01**] Raphaël Marvie, « *CODeX : proposition pour la description dynamique d'architectures à base de composants logiciels* », dans Actes des Journées composants : du système au langage, Besançon, France, octobre 2001.

[**Omg01b**] Raphaël Marvie et Philippe Merle, « *Holes in the CCM Specification* », OMG TC Document telecom/01-04-15, réunion commune des groupes de travail ORBOS et Telecom, meeting de l'*OMG*, Paris, France, avril 2001.

[**Cfse01**] Raphaël Marvie, Philippe Merle, Jean-Marc Geib et Mathieu Vadet, « *OpenCCM : une plate-forme ouverte pour composants CORBA* », Actes de la 2ème Conférence Française sur les Systèmes d'Exploitation (*CFSE'2*), pages 1 - 12, Paris, France, 24 - 26 avril 2001.

[**Ejndp01**] Raphaël Marvie, Philippe Merle, Jean-Marc Geib et Sylvain Leblanc, « *TORBA : vers des contrats de courtage* », *Electronic Journal on Network and Distributed Processing* (EJNDP), volume 1, num. 11, pages 1 - 18, Pau, France, mars 2001. ISSN : 1262-3261.

[**Isads01**] Raphaël Marvie, Philippe Merle, Jean-Marc Geib et Sylvain Leblanc, « *Type-safe Trading Proxies Using TORBA* », Actes de la 5th International Conference on Autonomous Distributed Systems (*ISADS'01*), pages 303 - 310, Dallas, Texas, Etats-Unis, 26 - 28 mars 2001. IEEE, ISBN : 0-7695-1065-5.

[**Coots01**] Raphaël Marvie, Philippe Merle, Jean-Marc Geib et Sylvain Leblanc, « *TORBA: Trading Contracts for CORBA* » Actes de la 6th USENIX Conference on Object-Oriented Technologies and Systems (*COOTS'01*), pages 1 - 14, San Antonio, Texas, Etats-Unis, 29 janvier - 02 février 2001. USENIX Associations, ISBN : 1-880446-12-X.

[**Hermes00b**] Raphaël Marvie, Philippe Merle, Jean-Marc Geib et Christophe Gransart, « *CCM + IDLscript = Applications Distribuées* », publié dans « *Evolution des plates-formes orientées objets répartis* », numéro spécial de la revue *Calculateurs Parallèles et Réseaux*, volume 12, num. 1, pages 75 - 104, Paris, France, 2000. Editions Hermès, ISBN : 2-7462-0169-0.

[**Hermes00a**] Marie-Claude Pellegrini, Olivier Potonniée, Raphaël Marvie, Sébastien Jean et Michel Riveil, « *CESURE : une plate-forme d'applications adaptables et sécurisées pour usagers mobiles* », publié dans « *Evolution des plates-formes orientées objets répartis* », numéro spécial de la revue *Calculateurs Parallèles et Réseaux*, volume 12, num. 1, pages 113 - 120, Paris, France, 2000. Editions Hermès, ISBN : 2-7462-0169-0.

[**Notere00**] Raphaël Marvie, Philippe Merle, Jean-Marc Geib et Sylvain Leblanc « *TORBA : vers des contrats de courtage* », Actes du 3ème Colloque International sur les NOuvelles TEchnologies de la REpartition (*NOTERE'2000*), pages 3 - 20, ENST 2000 S 003, Paris, France, 21 - 24 novembre 2000.

[**Cesure00d**] Guy Bernard, Raphaël Marvie, Erik Putricz et Chantal Taconnet, « *CESURE - Spécification de l'infrastructure système* », rapport technique du projet *RNRT CESURE* 98-5, novembre 2000.

[**Cesure00c**] Raphaël Marvie et Philippe Merle, « *Vers un modèle de composants pour CESURE* », rapport technique du projet *RNRT CESURE* 98-3, novembre 2000.

[**Doa00**] Raphaël Marvie, Philippe Merle et Jean-Marc Geib, « *Towards a Dynamic CORBA Component Platform* », Actes du 2nd International Symposium on Distributed Object Applications (*DOA'2000*), pages 305 - 314, Anvers, Belgique, 21 - 23 septembre 2000. IEEE, ISBN : 0-7695-0819-7.

[**Sigops00**] Raphaël Marvie, Marie-Claude Pellegrini et Olivier Potonniée, « *Smart Cards: A System Support for Service Accessibility from Heterogeneous Devices* », Actes du SIGOPS European Workshop 2000, Kolding, Danemark, Septembre 2000.

[**Ecoop00**] Raphaël Marvie et Philippe Merle, « *CORBA: From Objects to Components* », tutoriel à la 14th European Conference on Object Oriented Programming (*ECOOP'2000*), Sophia Antipolis et Cannes, France, 12 - 16 juin 2000.

[**Gdc00**] Raphaël Marvie, Marie-Claude Pellegrini, Olivier Potonniée et Sébastien Jean « *Value-added Services: How to Benefit from Smart Card* », Actes de la Gemplus Developer Conference 2000 (*GDC'2000*), Montpellier, France, juin 2000.

[**Ocm00**] Michel Riveil, Marie-Claude Pellegrini, Olivier Potonniée et Raphaël Marvie, « *Adaptabilité et disponibilité des applications pour les usagers mobiles* », Actes de la conférence Objets, Composants et Modèles (*OCM'2000*), EMN, Nantes, France, mai 2000.

[**Snpd00**] Raphaël Marvie, Philippe Merle et Jean-Marc Geib, « *A Dynamic Platform for CORBA Component-Based Applications* », Actes de la First International Conference on Software Engineering Applied to Networking and Parallel/Distributed Computing (*SNPD'00*), pages 352 - 357, Reims, France, 18 - 21 mai 2000. IACIS, ISBN : 0-9700776-0-2.

[**Cesure00b**] Raphaël Marvie et Marie-Claude Pellegrini, « *Etat de l'art des modèles de composants* », rapport technique du projet *RNRT CESURE* 98-2, mai 2000.

[**Cesure00a**] Roland Balter, Guy Bernard, Jean-Marc Geib, Daniel Hagimont, Raphaël Marvie, Philippe Merle, Pierre Paradinas, Marie-Claude Pellegrini, Olivier Potonniée, Erik Putricz et Chantal Taconnet, « *CESURE - Spécifications fonctionnelles* », rapport technique du projet *RNRT CESURE* 98-1, mai 2000.

[**Middleware00**] Raphaël Marvie, Philippe Merle et Jean-Marc Geib, « *A Dynamic Platform for CORBA Component-Based Applications* », Session poster IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (*Middleware 2000*), Palissades, NY, Etats-Unis. avril 2000.

[**Icssea99**] Raphaël Marvie, Philippe Merle, Jean-Marc Geib et Christophe Gransart, « *Des objets aux composants CORBA, première étude et expérimentation avec CorbaScript* », Actes de la 12th International Conference on Software and Systems Engineering and

their Applications (*ICSSEA'99*), volume 12, papier 11 - 4, Paris, France, 8 - 10 décembre 1999.

[Cnes99] Raphaël Marvie et Philippe Merle, « *Un rapide ;-) tour d'horizon des composants CORBA 3.0* », atelier Middleware organisé par le *CNES*, Toulouse, France, 16 - 17 novembre 1999.

[Lifl99] Raphaël Marvie, « *Synthèse sur la proposition de l'OMG pour les composants CORBA* », rapport technique LIFL, juillet 1999.

[Objet99] Raphaël Marvie, Philippe Merle, Jean-Marc Geib et Christophe Gransart, « *CorbaScript : un langage de script pour CORBA* », tutoriel à la conférence *Objet 99*, Nantes, France, 20 mai 1999.

Annexe B

Acronymes

La signification des acronymes utilisés dans ce documents est, en règle générale, précisée lors de leur première utilisation dans chaque chapitre. Cette annexe regroupe tous ces acronymes, leur signification en anglais et une équivalence en français.

ACN *Architecture Construction Notation*, Notation de construction d'architectures.

ADN *Architecture Description Notation*, Notation de description d'architectures.

ADL *Architecture Description Language*, Langage de description d'architecture.

AOP *Aspect Oriented Programming*, Programmation orientée aspects.

API *Application Programming Interface*, Interface de programmation des applications.

CAD *Component Assembly Descriptor*, Descripteur d'assemblage de composants.

CCD *CORBA Component Descriptor*, Descripteur de composant CORBA.

CCM *CORBA Component Model*, Modèle de composants CORBA.

CIDL *Component Implementation Definition Language*, Langage de définition des implantation de composants.

CIF *Component Implementation Framework*, Cadre d'implantation des composants.

COM *Component Object Model*, Modèle de composants objet.

CORBA *Common Object Request Broker and Architecture*, Architecture commune pour bus d'invocation de requêtes d'objets.

COTS *Commercial Off The Shelf*, Logiciel sur l'étagère.

CPF *Component Property File*, Fichier de propriétés de composant.

CWM *Common Warehouse Metamodel*, Méta-modèle commun de stockage de données.

DCOM *Distributed Component Object Model*, Modèle de composants distribués objet.

DOM *Document Object Model*, Modèle de document objet.

DSTC *Distributed Systems Technologies Center.*

DTD *Data Type Definition*, Définition de type de données.

EDOC *Enterprise Distributed Object Computing*, Informatique d'entreprise à base d'objets distribués.

EJB *Enterprise JavaBeans*, Composants Java pour l'entreprise.

IDL *Interface Definition Language*, Langage de définition d'interfaces.

IDN *Interface Definition Notation*, Notation de définition d'interfaces.

ISO *International Standardization Organization*, Organisation internationale de standardisation.

ITU *International Telecommunication Union*, Union internationale des télécommunications.
J2EE *Java 2 Enterprise Edition*, Java 2 édition entreprises.
JNDI *Java Naming and Directory Interface*, Interface Java de nommage et de d'annuaire.
JVM *Java Virtual Machine*, Machine Virtuelle Java.
MDA *Model Driven Architecture*, Architecture dirigée par les modèles.
MOF *Meta Object Facility*, Ressource pour les méta-objets.
- **M0** Niveau des instances de la pile de méta-modélisation MOF.
- **M1** Niveau des modèles de la pile de méta-modélisation MOF.
- **M2** Niveau des méta-modèles de la pile de méta-modélisation MOF.
- **M3** Niveau du méta-méta-modèles de la pile de méta-modélisation MOF.

MTS *Microsoft Transaction Server*, Serveur de Transactions Microsoft.
OCL *Olan Configuration Language*, Langage de configuration Olan.
ODP *Open Distributed Processing*, Traitements informatiques ouverts distribués.
OMG *Object Management Group.*
OSD *Open Software Descriptor*, Descripteur ouvert de logiciel.
PIM *Platform Independent Model*, Modèle indépendant des plates-formes.
PSM *Platform Specific Model*, Modèle spécifique aux plates-formes.
RMI *Remote Method Invocation*, Invocation de méthodes à distance.
RUP *Rational Unified Process*, Processus Unifié de Rational.
SoC *Separation of Concerns*, Séparation des préoccupations.
SOP *Subject Oriented Programming*, Programmation orientée sujets.
UML *Unified Modeling Language*, Langage de modélisation unifié.
XMI *XML Metadata Interchange (Format)*, Format XML d'échange de méta-données.
XML *eXtensible Markup Language*, Langage de balises extensible.

Bibliographie

[1] M. Aksit, K. Wakita, J. Bosch, et L. Bergmans. « Abstracting Object Interactions Using Composition Filters ». volume 791, pages 152–184, 1994.

[2] J. Aldrich, C. Chambers, et D. Notkin. « Architectural Reasoning in ArchJava ». Dans *Proceedings of the 16th European Conference on Object Oriented Programming (ECOOP'2002)*, Malaga, Espagne, Juin 2002.

[3] J. Aldrich, C. Chambers, et D. Notkin. « ArchJava : Connecting Software Architecture to Implementation ». Dans *Proceedings of the 2002 International Conference on Software Engineering (ICSE'2002)*, 2002.

[4] R. Allen. « *A Formal Approach to Software Architecture* ». Thèse de Doctorat, School of Computer Science, Carnegie Mellon University, Pittsburg, Mai 1997.

[5] R. Allen et D. Garlan. « Formalizing Archietctural Connection ». Dans *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, Sorrento, Italie, Mai 1994.

[6] R. Balter, L. Bellissard, F. Boyer, M. Riveill, et J.-Y. Vion-Dury. « Architecturing and Configuring Distributed Applications with Olan ». Dans *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, Angleterre, Septembre 1998.

[7] L. Bellisard. « *Construction et configuration d'applications réparties* ». Thèse de Doctorat, Institut National Polytechnique de Grenoble, 1997.

[8] L. Bellisard et M. Riveill. « From Distributed Objects to Distributed Components: The Olan Approach ». Dans *Workshop Putting Distributed Objects to Work, ECOOP'96*, Autriche, Juillet 1996.

[9] A. Birrell et B. Nelson. « Implementing Remote Procedure Call ». Rapport Technique CSL-83-7, Xerox, Octobre 1983.

[10] X. Blanc. « *Echange de spécifications hétérogènes et réparties* ». Thèse de Doctorat, Laboratoire d'Informatique de Paris 6, Octobre 2001.

[11] E. Bruneton. « *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties* ». Thèse de Doctorat, Institut National Polytechnique de Grenoble, Octobre 2001.

[12] E. Bruneton et M. Riveill. « JavaPod : une plate-forme à composants adaptable et extensible ». Rapport Technique 3850, INRIA, Janvier 2000.

[13] J. Bézivin. « From Objects to Model Transformation with the MDA ». Dans *Proceedings of TOOLS'USA*, volume IEEE-TOOLS 39, Santa Barbara, Août 2001.

[14] J. Bézivin et O. Gerbé. « Towards a Precise Definition of the OMG/MDA Framework ». Dans *Proceedings of the Conference on Autonomous Software Egineering (ASE'01)*, San Diego, CA, Etats-Unis, Novembre 2001.

[15] J. Bézivin, J. Lanneluc, et R. Lemesle. « sNets: The Core Formalism for an Object-Oriented Case Tool ». *Object-Oriented Technology for Databases and Software Systems*, pages 224–239, 1995. World Scientific Publishers.

[16] J. Bézivin et R. Lemesle. « Some Initial Considerations on the Layered Organization of Metamodels ». Université de Nantes.

[17] Corbaweb. Site web de jIdlscript. URL: http://corbaweb.lifl.fr/jidlscript/.

[18] S. Crawley, S. Davis, J. Indulska, S. McBride, et K.Raymond. « Meta-meta is better-better! ». Dans *Proceedings of the IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, Cottbus, Allemagne, Septembre 1997.

[19] K. Czarnecki et U.W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[20] E. Dashofy. « Issues in Generating Data Bindings for an XML Schema-Based Language ». Dans *Proceedings of the Workshop on XML Technologies and Software Engineering (XSE2001)*, Toronto, Canada, 2001.

[21] E. Dashofy et A. van der Hoek. « Representing Product Family Architectures in an Extensible Architecture Description Language ». Dans *Proceedings of the International Workshop on Product Family Engineering (PFE-4)*, Bilbao, Espagne, Octobre 2001.

[22] E. Dashofy, A. van der Hoek, et R. Taylor. « A Highly-Extensible, XML-Based Architecture Description Language ». Dans *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands, 2001.

[23] N. de Palma, L. Bellissard, et M. Riveill. « Dynamic Reconfiguration of Agent-Based Applications ». Dans *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madère, Portugal, Avril 1999.

[24] L. DeMichiel, L. Yalçinalp, et S. Krishnan. « *Enterprise Java Beans Specification Version 2.0 - Public Draft* ». Sun Microsystems, Mai 2000.

[25] DSTC. « *dMOF User Guide release 1.0* ». Distributed Systems Technology Center, 2000. *http://www.dstc.edu.au*.

[26] F. Duclos. « *Environnement de gestion de services non-fonctionnels dans les applications à composants* ». Thèse de Doctorat, Université Joseph Fourrier, Grenoble, 2002.

[27] W. Ellis, R. Hilliard, P. Poon, D. Rayford, T. Saunders, B. Sherlund, et R. Wade. « Towards a Recommanded Pratice for Architecture Description ». Dans *Proceedings 2nd International Conference on Engineering of Complex Computer Systems*, Montréal, Canada, Octobre 1996. IEEE.

[28] J. Ernst. « *Introduction to CIDF* ». Electronic Industries Association, Integrated Systems, Inc., 1997.

[29] E. Gamma, R. Helm, R. Johnson, J. Vlissides, et G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Westley Professional Computing, USA, 1995.

[30] D. Garlan, R. Allen, et J. Ockerbloom. « Architectural Mismatch or Why it's hard to build systems out of existings parts ». Dans *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, Seattle, WA, Etats-Unis, 1995.

[31] D. Garlan, R. Monroe, et D. Wile. « Acme: An Architecture Description Interchange Language ». Dans *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, Novembre 1997.

[32] J.-M Geib, C. Gransart, et P. Merle. *CORBA : des concepts à la pratique*. Ed. Dunod, Paris, 1999. *ISBN: 2-10-004806-6*.

[33] O. Gerbé et B. Kerhervé. « Modeling and Metamodeling Requirements for Knowledge Management ». Dans *Proceedings of OOPSLA Workshop on Model Engineering with CIDF*, Vancouver, Canada, Octobre 1998.

[34] R. Grimes. *Professional DCOM Programming*. Wrox Press ltd., Birmingham, Canada, 1997.

[35] N. Guarino et C. Welty. « Towards a Methodology for Ontology Based Model Engineering ». Dans *International Workshop on Model Engineering (in conjunction with ECOOP 2000)*, Nice, France, Juin 2000.

[36] G. Hamilton. « *Java Beans Specification v1.01* ». Sun Microsystems, Juillet 1997.

[37] W. Harrison et H. Ossher. « Subject-oriented programming (A critique of pure objects) ». Dans *Proceedings of OOPSLA'93, volume 28 of SIGPLAN Notices*, pages 411–428, octobre 1993.

[38] IS0. « *Open Distributed Processing Reference Model – parts 1-4* ». International Standard Organization, 1995. ISO 10746-1..4.

[39] I. Jacobson, G. Booch, et J. Rumbaugh. *Le processus unifié de développement logiciel*. Eyrolles, traduction française edition, 2000. *ISBN: 2-212-09142-7*.

[40] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, et W. Griswold. « An Overview of AspectJ ». 2001. AspectJ white paper submitted to the the European Conference on Object-Oriented Programming (ECOOP'01).

[41] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, et J. Irwin. « Aspect-Oriented Programming ». Dans *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 de *Lecture Notes in Computer Science*, pages 220–242. Springer, juin 1997.

[42] J. Kramer et J. Magee. « The Evolving Philosophers Problem : Dynamic Change Management ». *IEEE Trans. on Software Engineering*, 16(11):1293–1306, Novembre 1990.

[43] S. Leblanc, R. Marvie, P. Merle, et J.-M. Geib. « *Les intergiciels, développements récents dans CORBA, Java RMI et les agents mobiles* », chapitre TORBA : contrats de courtage pour CORBA, pages 47–72. Hermes science, Avril 2002. *ISBN: 2-7462-0432-0*.

[44] K. Lieberherr, D. Lorenz, et M. Mezini. « Programming with aspectual components ». Rapport Technique Technical Report NU-CCS-99-01, Northeastern University's College of Computer Science, apr 1999.

[45] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. Addison-Wesley,, 2000. ISBN 0-534-94602-X.

[46] C. Lopes et W. Hursch. « Separation of Concerns ». Rapport technique, College of Computer Science, Northeastern University, Boston, MA, Etats-Unis, Février 1995.

[47] J. Magee, N. Dulay, et J. Kramer. « A Constructive Development Environment for Parallel and Distributed Programs ». Dans *Proceedings of the International Workshop on Configurable Distributed Systems*, Pittsburg, Etats-Unis, Mars 1994.

[48] R. Marvie, P. Merle, et O. Caron. « Le modèle de composants CCM ». Rapport Technique 1.1 du projet ACCORD, LIFL, Avril 2002.

[49] R. Marvie, P. Merle, et J.-M. Geib. « Towards a Dynamic CORBA Component Platform ». Dans *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, pages 305–314, Anvers, Belgique, Septembre 2000. IEEE. *ISBN: 0-7695-0819-7.*

[50] R. Marvie, P. Merle, J.-M. Geib, et C. Gransart. « CCM + IDLscript = Applications Distribuées ». *Evolution des plates-formes orientées objets répartis, numéro spécial de Calculateurs Parallèles et Réseaux*, 12(1):75–104, 2000. Ed. Hermès, *ISBN: 2-7462-0169-0.*

[51] R. Marvie, P. Merle, J.-M. Geib, et S. Leblanc. « TORBA: vers des contrats de courtage ». Dans *3ème Colloque International sur les NOuvelles TEchnologies de la RÉpartition (NOTERE'2000)*, pages 3–20, Paris, France, Novembre 2000.

[52] R. Marvie, P. Merle, J.-M. Geib, et S. Leblanc. « TORBA: Trading Contracts for CORBA ». Dans *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, pages 1–14, San Antonio, Texas, USA, Janvier 2001. USENIX. *ISBN: 1-880446-12-X.*

[53] R. Marvie, P. Merle, J.-M. Geib, et M. Vadet. « OpenCCM: une plate-forme ouverte pour composants CORBA ». Dans *Actes de la 2ème Conférence Française sur les Systèmes d'Exploitation (CFSE'2)*, pages 1–12, Paris, France, Avril 2001.

[54] R. Marvie, M.-C. Pellegrini, et O. Potonniée. « Smart Cards: A System Support for Service Accessibility from Heterogeneous Devices ». Dans *Proceedings of the SIGOPS European Workshop 2000*, Kolding, Danemark, Septembre 2000.

[55] V. Matena et M. Hapner. « *Enterprise Java Beans Specification v1.1 - Final Release* ». Sun Microsystems, Mai 1999.

[56] N. Medvidovic et R. Taylor. « A Framework for Classifying and Comparing Architecture Description Languages ». Dans M. Jazayeri et H. Schauer, editeurs, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer–Verlag, 1997.

[57] N. Medvidovic, R. Taylor, et E. Whitehead. « Formal Modeling of Software Architectures at Multiple Levels of Abstraction ». Dans *Proceedings of the California Software Symposium*, pages 16–27, Los Angeles, CA, Avril 1986.

[58] P. Merle. « *CorbaScript - CorbaWeb : propositions pour l'accès à des objets et services distribués* ». Thèse de Doctorat, Laboratoire d'Informatique Fondamentale de Lille, Villeneuve d'ascq, France, Janvier 1997.

[59] P. Merle, C. Gransart, et J.-M Geib. « CorbaScript and CorbaWeb: A Generic Object Oriented Dynamic Environment upon CORBA ». Dans *Proceedings of TOOLS Europe 96*, Paris, France, Juin 1996.

[60] P. Merle, C. Gransart, et J.-M. Geib. « Using and Implementing CORBA Objects with CorbaScript ». *Object-Oriented Parallel and Distributed Programming*, 2000. Ed. Hermes, *ISBN: 2-7462-0091-0.*

[61] Metamodel.com. Site Web www.metamodel.com. *http://www.metamodel.com.*

[62] M. Mezini et K. Lieberherr. « Adaptative plug-and-play components for evolutionary software development ». *SIGPLAN Notices*, 33:96–116, 1998. In Proceedings of OOPSLA'98, ACM Press.

[63] M. Mezini, L. Seiter, et K. Lieberherr. « *Component integration with pluggable composite adapters* », chapitre Software Architectures and Component Technology: The State of the Art in Research and Practice. In L. Bergmans and M. Aksit, kluwer academic publishers edition, 2001.

[64] OMG. « *UML Profile for Enterprise Distributed Computing, Request For Proposal* ». Object Management Group, Octobre 1999. OMG TC Document ad/1999-03-10.

[65] OMG. « *Meta Object Facility (MOF) Specification, Version 1.3* ». Object Management Group, Mars 2000. OMG TC Document formal/00-04-03.

[66] OMG. « *UML Profile for CORBA* ». Object Management Group, Octobre 2000. OMG TC Document ptc/2000-10-01.

[67] OMG. « *Common Warehouse Metamodel (CWM) Specification, Volume 1* ». Object Management Group, Octobre 2001. OMG TC Document formal/2001-10-01.

[68] OMG. « *Common Warehouse Metamodel (CWM) Specification, Volume 2* ». Object Management Group, Octobre 2001. OMG TC Document formal/2001-10-27.

[69] OMG. « *CORBA 3.0 New Components Chapters* ». Object Management Group, Décembre 2001. OMG TC Document ptc/2001-11-03.

[70] OMG. « *CORBA Scripting Language Specification, v1.0* ». Object Management Group, Juin 2001. OMG TC Document formal/01-06-05.

[71] OMG. « *CORBA/IIOP 2.4.2 Specification* ». Object Management Group, Février 2001. OMG TC Document formal/01-02-01.

[72] OMG. « *Model Driven Architecture (MDA)* ». Object Management Group, Juillet 2001. OMG TC Document ormsc/2001-07-01.

[73] OMG. « *OMG Unified Modeling Language Specification* ». Object Management Group, September 2001. OMG TC Document formal/2001-09-67.

[74] OMG. « *OMG Metadata Interchange (XMI) Specification, Version 1.2* ». Object Management Group, Janvier 2002. OMG TC Document formal/02-01-01.

[75] H. Ossher, K. Kaplan, W. Harrison, A. Matz, et V. Kruskal. « Subject-Oriented Composition Rules ». Dans *Proceedings of OOPSLA'95*, volume 30 de *Sigplan Notices*, pages 235–250. ACM Press, 1995.

[76] H. Ossher et P. Tarr. « Multi-dimensional Separation of Concerns in Hyperspace ». Rapport Technique RC 21452(96717)16APR99, IBM T.J. Watson Research Center, Avril 1999.

[77] H. Ossher et P. Tarr. « *Multi-dimensional separation of concerns and the hyperspace approach* », chapitre Software Architectures and Component Technology: The State of the Art in Research and Practice. In L. Bergmans and M. Aksit, kluwer academic publishers edition, 2001.

[78] X. Le Pallec. MOF, Workflow et travail collaboratif. Présentation du groupe Méta, Avril 2000. *http://www.lip6.fr/meta*.

[79] X. Le Pallec et G. Bourguin. « RAM3 : un outil dynamique pour le Meta-Object Facility ». Dans *Actes de la conférence Langages et Modèles à Objets (LMO'01)*, pages 74–94, Le Croisic, France, Janvier 2001.

[80] N. De Palma. « ... ». Thèse de Doctorat, Université Joseph Fournier, Grenoble, Octobre 2001.

[81] D. Parnas. « On the criteria to be used in decomposing systems in modules ». *Communication on the ACM*, 15(12):1053–1058, 1972.

[82] M.-C. Pellegrini, O. Potonniée, R. Marvie, S. Jean, et M. Riveill. « CESURE : une plate-forme d'applications adaptables et sécurisées pour usagers mobiles ». *Evolution des plate-formes orientées objets répartis, numéro spécial de Calculateurs Parallèles et Réseaux*, 2000. Ed. Hermès, *ISBN: 2-7462-0169-0*.

[83] J. Poole. « Model-Driven Architecture: Visions, Standards, and Emerging Technologies ». Dans *Workshop on Metamodeling and Adaptive Object Models, ECOOP 2001*, Juin 2001.

[84] M. Riveill, M.-C. Pellegrini, O. Potonniée, et R. Marvie. « Adaptabilité et disponibilité des applications pour des usagers mobiles ». Dans *Objets, Composants et Modèles (OCM'2000)*, Nantes, France, Mai 2000.

[85] D. Rogerson. *Inside COM*. Microsoft Press, Redmond VA, USA, 1997.

[86] M. Saeki. « Toward Formal Semantics of Meta-Models ». Dans *Proceedings of the Workshop on ... (ECOOP 2000)*, Nice, France, Juin 2000.

[87] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Toung, et G. Zelesnik. « Abstraction for Software Architecture and Tools to Support Them ». *IEEE Trans. Software Engineering*, SE-21(4):314–335, Avril 1995.

[88] M. Sibilla et F. Jocteur Monrozier. « SUMO: Supervision globale des systèmes ». Dans *Actes de l'atelier Middleware et les applications internet*, Toulouse, France, Novembre 1999. Centre National d'Etudes Spaciales.

[89] R. Smith et D. Ungar. « Programming as an Experience: The Inspiration for Self ». Dans *ECOOP'95 Conference Proceedings*, Danemark, Août 1995.

[90] Sun Microsystems. « *Java 2 Enterprise Edition Home Page* ». *http://java.sun.com/j2ee/*.

[91] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Westley, 1999. *ISBN: 0-201-17888-5*.

[92] A. Tannenbaum. *Metadata Solutions - Using Metamodels, Repositories, XML, and Enterprise Portals to Generate Information on Demand*. Addison Westley, 2002.

[93] P. Tarr, H. Ossher, W. Harrison, et S. Sutton. « N degrees of separation: Multi-dimensional separation of concerns ». Dans *Proceedings of the International Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.

[94] P. Tarr, H. Ossher, et S.M. Sutton. « N Degrees of Separation: Multi-dimensional Separation of Concerns ». Dans *Proceedings of the International Conference on Software Engineering (ICSE'99)*, Mai 1999.

[95] R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead, J. Robbins, K. Nies, P. Oreizy, et D. Dubrow. « A Component and Message-Based Architectural Style for GUI Software ». *IEEE Transactions on Software Engineering*, 22(6):390–406, Juin 1996.

[96] Rapide Design Team. « *Rapide 1.0 Language Reference Manuals* ». Program Analysis and Verification Group, Computer System Lab, Stanford University, juillet 1997.

[97] T. Thai et H. Lam. *.Net Framework Essentials*. O'Reilly, 2001.

[98] D. Ungar et R. Smith. « Self: The Power of Simplicity ». *Lisp and Symbolic Computation*, 4(3):187–205, Juillet 1991.

[99] G. Vanwormhoudt, N. Carré, et L. Debrauwer. « Programmation par objets et contextes fonctionnels. Application de CROME à Smalltalk ». *L'Objet*, 3(4), 1998. Ed. Hermès.

[100] G. Vanwormout. « *CROME : un cadre de programmation par objets structurés en contextes* ». Thèse de Doctorat, Laboratoire d'Informatique Fondamentale de Lille, Villeneuve d'Ascq, Septembre 1999.

[101] W3C. « *Document Object Model (DOM) Level 2 Core Specification* ». World Wide Web Consortium, Novembre 2000. `http://www.w3.org/DOM/DOMTR`.

[102] W3C. « *XML Schema Part 0: Primer* ». World Wide Web Consortium, Mai 2001. `http://www.w3c.org/TR/xmlschema-0/`.

[103] W3C. « *XML Schema Part 1: Structures* ». World Wide Web Consortium, Mai 2001. `http://www.w3c.org/TR/xmlschema-1/`.

[104] W3C. « *XML Schema Part 2: Datatypes* ». World Wide Web Consortium, Mai 2001. `http://www.w3c.org/TR/xmlschema-2/`.

[105] N. Wang, D. Schmidt, M. Kircher, et K. Parameswaran. « Applying Reflective Techniques to Optimize a QoS-enabled CORBA Component Model Implementation ». Dans *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC 2000)*, Tapei, Taiwan, Octobre 2000.

[106] N. Wang, D. Schmidt, et D. Levine. Optimizing the CCM for High-Performance and Real-Time Applications. Middleware'2000, Work-in-Progress Session, 1999. New-York, USA.

Résumé

Les architectures logicielles sont centrales vis-à-vis des processus logiciels de construction des applications. Les langages de description d'architectures (ADLs) représentent la solution la plus adaptée pour définir de telles architectures. Toutefois, les concepts de ces ADLs sont en règle générale figés et leurs utilisations ne répond pas nécessairement aux différentes préoccupations des processus logiciels. De plus, leur aspect syntaxique ne facilite pas la coopération des différents acteurs de processus logiciels.

Notre proposition, CODeX, fournit un cadre de travail pour méta-modéliser des ADLs. Ces méta-modèles permettent de définir les concepts liés aux architectures nécessaires à un processus logiciel. Dans le but de structurer les ADLs ainsi définis et de permettre la collaboration des différents acteurs d'un processus logiciel, les méta-modèles sont définis en mettant en œuvre la séparation des préoccupations architecturales d'un processus logiciel. Cette séparation définit l'organisation des différents concepts d'un ADL.

Un outillage complément de ce cadre de travail permet de produire automatiquement, sur la base du méta-modèle d'un ADL, l'environnement associé pour la manipulation des architectures logicielles. Ce dernier est architecturé autour d'une représentation réifiée des architectures logicielles. Cette représentation suit la séparation des préoccupations définie dans le méta-modèle et permet d'offrir à chaque acteur d'un processus logiciel une vue dédiée sur l'architecture d'une application.

Mots-clé Méta-modélisation, séparation des préoccupations, langages de description d'architectures, composants logiciels, processus d'ingénierie du logiciel.

Abstract

Software architectures are centric regarding software engineering processes. Architecture Description Languages (ADLs) represent the most suited mean to define such architectures. Nevertheless, those ADLs concepts are most often fixed and their use does not always match the various concerns of software engineering processes. Moreover, their syntaxic reduced means do not ease the collaboration of the various actors of a software engineering process.

Our proposal (CODeX) provides a framework for ADL meta-modeling. A meta-model defines the architectural concepts required by a software engineering process. In order to structure such definitions and to improve the collaboration of the various actors, meta-models are defined following the sepration of software engineering processes' architectural concerns. This separation also defines the relationship between the various concepts of an ADL.

Together with this framework, a set of tools relying on the meta-model of an ADL automatically generate the associated environment to define and manipulate software architectures. The later is designed around a reified representation of software architectures. This representation also follows the separation of concerns defined in the meta-model and provides each actor a dedicated view on a software architecture.

Keywords Metamodeling, Separation of Concerns, Architecture Description Languages, Software Components, Software Engineering Process.